深入浅出MFC

wizardforcel

Published
with GitBook



目錄

深入浅出MFC 第二版

这个文档是从侯捷网站提供的繁体板简体化过来的。由于排版问题，有些繁体的术语在换行时候没有被替换，所以遇到问题大家可以对照原文比较一下。

《深入浅出ＭＦＣ》2/e电子书开放自由下载声明

致亲爱的大陆读者

我是侯捷（侯俊杰）。自从华中理工大学于1998/04出版了我的《深入浅出MFC》1/e简体版（易名《深入浅出WindowsMFC程序设计》）之后，陆陆续续我收到了许多许多的大陆读者来函。其中对我的赞美、感谢、关怀、殷殷垂询，让我非常感动。

《深入浅出MFC》2/e早已于1998/05于台湾出版。之所以迟迟没有授权给大陆进行简体翻译，原因我曾于回复读者的时候说过很多遍。我在此再说一次。

1998年中，本书之发行公司松岗（UNALIS）即希望我授权简体版，然因当时我已在构思3/e，预判3/e繁体版出版时，2/e简体版恐怕还未能完成。老是让大陆读者慢一步看到我的书，令我至感难过，所以便请松岗公司不要进行2/e简体版之授权，直接等3/e出版后再动作。没想到一拖经年，我的3/e写作计划并没有如期完成，致使大陆读者反而没有《深入浅出MFC》2/e简体版可看。

《深入浅出MFC》3/e没有如期完成的原因是，MFC本体架构并没有什么大改变。《深入浅出MFC》2/e书中所论之工具及程序代码虽采用VC5+MFC42，仍适用于目前的VC6+MFC421（唯，工具之画面或功能可能有些微变化）。

由于《深入浅出MFC》2/e并无简体版，因此我时时收到大陆读者来信询问购买繁体版之管道。一来我不知道是否台湾出版公司有提供海外邮购或电购，二来即使有，想必带给大家很大的麻烦，三来两岸消费水平之差异带给大陆读者的负担，亦令我深感不安。

因此，此书虽已出版两年，鉴于仍具阅读与技术上的价值，鉴于繁简转译制作上的费时费工，鉴于我对同胞的感情，我决定开放此书内容，供各位免费阅读。我已为《深入浅出MFC》2/e 制作了PDF 格式之电子文件，放在 http://www.jjhou.com 供自由下载。北京 http://expert.csdn.net/jjhou 有侯捷网站的一个GBK mirror，各位也可试着自该处下载。

我所做的这份电子书是繁体版，我没有精力与时间将它转为简体。这已是我能为各位尽力的极限。如果（万一）您看不到文件内容，可能与字形的安装有关-虽然我已尝试内嵌字形。anyway，阅读方面的问题我亦没有精力与时间为您解决。请各位自行开辟讨论区，彼此交换阅读此电子书的solution。请热心的读者告诉我您阅读成功与否，以及网上讨论区（如有的话）在哪里。 曾有读者告诉我，《深入浅出MFC》1/e

简体版在大陆被扫描上网。亦有读者告诉我，大陆某些书籍明显对本书侵权（详细情况我不清楚）。这种不尊重作者的行为，我虽感遗憾，并没有太大的震惊或难过。一个社会的进化，终究是一步一步衍化而来。台湾也曾经走过相同的阶段。但盼所有华人，尤其是我们从事智能财产行为者，都能够尽快走过灰暗的一面。

在现代科技的协助下，文件影印、文件复制如此方便，智财权之尊重有如「君子不欺暗室」。没有人知道我们私下的行为，只有我们自己心知肚明。《深入浅出MFC》2/e 虽免费供大家阅读，但此种作法实非长久之计。为计久长，我们应该尊重作家、尊重智财，以良好（至少不差）的环境培养有实力的优秀技术作家，如此才有源源不断的好书可看。

我的近况，我的作品，我的计划，各位可从前述两个网址获得。欢迎各位写信给我（jjhou@ccca.nctu.edu.tw）。虽然不一定能够每封来函都回复，但是我乐于知道读者的任何点点滴滴。

关于《深入浅出ＭＦＣ》２／ｅ电子书

《深入浅出ＭＦＣ》２／ｅ电子书共有五个档案：

档名	內容	大小ｂｙｔｅｓ
dissecting MFC 2/e part1.pdf	chap1~chap3	3,384,209
dissecting MFC 2/e part2.pdf	chap4	2,448,990
dissecting MFC 2/epart3.pdf	chap5~chap7	2,158,594
dissecting MFC 2/e part4.pdf	chap8~chap16	5,171,266
dissecting MFC 2/e part5.pdf	appendixA,B,C,D	1,527,111

每个档案都可个别阅读。每个档案都有书签（亦即目录连接）。每个档案都不需密码即可打开、选择文字、打印。

请告诉我您的资料

每一位下载此份电子书的朋友，我希望您写一封email給我（jjhou@ccca.nctu.edu.tw），告诉我您的以下資料，俾让我对我的读者有一些基本瞭解，谢谢。

姓名： 现职： 毕业学校科系： 年龄： 性别： 居住省份（如是台湾读者，请写县市）： 对侯捷的建议：

--theend

第 0 章 你一定要知道（导读）

这本书适合谁

深入浅出MFC是一本介绍MFC（Microsoft Foundation Classes）程序设计技术的书籍。对于Windows 应用软件的开发感到兴趣，并欲使用Visual C++ 整合环境的视觉开发工具，以MFC为程序基础的人，都可以从此书获得最根本最重要的知识与实例。

如果你是一位对Application Framework和面向对象（Object Oriented）观念感兴趣的技术狂热份子，想知道神秘的Runtime Type Information、Dynamic Creation、Persistence、Message Mapping 以及Command Routing 如何实现，本书能够充分满足你。事实上，依我之见，这些核心技术与彻底学会操控MFC乃同一件事情。

全书分为四篇：

第一篇【勿在浮砂筑高台】提供进入 MFC 核心技术以及应用技术之前的所有技术基础，包括：

Win32 程序观念 ：message based,event driven,multitasking, multithreading, console programming。

- C++ 重要技术：类与对象、this 指针与继承、静态成员、虚函数与多态、模板（template）类、异常处理（exception handling）。
- MFC 六大技术之简化仿真（Console 程序）

第二篇【欲善工事先利其器】提供给对Visual C++ 整合环境全然陌生的朋友一个导引。这一篇当然不能取代 Visual C++ User's Guide 的地位，但对整个软件开发环境有全盘以及概观性的介绍，可以让初学者迅速了解手上掌握的工具，以及它们的主要功能。

第三篇【浅出 MFC 程序设计】介绍一个 MFC 程序的生死因果。已经有 MFC 程序经验的朋友，不见得不会对本篇感到惊艳。根据我的了解，太多人使用 MFC 是「只知道这么做，不知道为什么」；本篇详细解释 MFC 程序之来龙去脉，为初入 MFC 领域的读者奠定扎实的基础。说不定本篇会让你有醍醐灌顶之感。

第四篇【深入 MFC 程序设计】介绍各式各样 MFC 技术。「只知其然 不知其所以然」的不良副作用，在程序设计的企图进一步开展之后，愈来愈严重，最终会行不得也！那些最困扰我们的 MFC 宏、MFC 常数定义，不得一窥堂奥的 MFC 黑箱作业，在本篇陆续曝光。本篇将使您高喊：Eureka！

阿基米得在洗澡时发现浮力原理，高兴得来不及穿上裤子，跑到街上大喊：Eureka（我找到了）。

范例程序方面，第三章有数个Console程序（DOS-like 程序，在Windows系统的DOS Box 中执行），模拟并简化Application Framework 六大核心技术。另外，全书以一个循序渐进的Scribble 程序（Visual C++ 所附范例），从第七章开始，分章探讨每一个MFC应用技术主题。第13章另有三个程序，示范 Multi-View 和 Multi-Document 的情况。

14章~16 章是第二版新增内容，主题分别是MFC 多线程程序设计、Custom AppWizard、以及如何使用Component Gallery 提供的ActiveX controls 和components。

你需要什么技术基础

从什么技术层面切入Windows软件开发领域？C/SDK？抑或C++/MFC？这一直是个引起争议的论题。就我个人观点，C++/MFC 程序设计必须跨越四大技术障碍：

1. 面向对象观念与C++ 语言。

2. Windows 程序基本观念（程序进入点、消息流动、窗口函数、callback...）。

3. Microsoft Foundation Classes（MFC）本身。

4. Visual C++ 整合环境与各种开发工具（难度不高，但需熟练）。

换言之，如果你从未接触C++，千万不要阅读本书，那只会打击你学习新技术的信心而已。如果已接触过C++ 但不十分熟悉，你可以一边复习C++ 一边学习MFC，这也是我所鼓励的方式（很多人是为了使用MFC 而去学习C++ 的）。C++ 语言的继承（inheritance）特性对于我们使用MFC尤为重要，因为使用MFC 就是要继承各个类并为己用。所以，你应该对C++ 的继承特质（以及虚函数，当然）多加体会。我在第 2 章安排了一些C++ 的必要基础。我所挑选的题目都是本书会用到的技术，而其深度你不见得能够在一般 C++ 书籍中发现。

如果你有C++ 语言基础，但从未接触过Win16 或 Win32 程序设计，只在 DOS 环境下开发过软件，我在第 1 章为你安排了一些 Win32 程序设计基础。这个基础至为重要，只会在各个Wizards上按来按去，却不懂所谓 message loop 与 window procedure 的人，不可能搞定Windows 程序设计——不管你用的是MFC或OWL或Open Class Library，不管你用的是Visual C++或Borland C++或VisualAge C++。

名词界定：

API—Application Programming Interface** 系统开放出来，给程序员使用的接口，就是API。一般人的观念中API是指像 C 函数那样的东西，不尽然！DOS 的中断向量（interrupt vector）也可以说是一种API，OLE Interface（以 C++ 类的形式呈现）也可以说是一种 API。不是有人这么说吗：MFC 势将成为 Windows 环境上标准的 C++ API（我个人认为这句话已成为事实）。

SDK**—Software Development Kit** 原指软件开发工具。每一套环境都可能有自己的SDK，例如Phar Lap的386|DOS Extender 也有自己的SDK。在Windows这一领域，SDK原是指Microsoft 的软件开发工具，但现在已经变成一个一般性名词。凡以Windows raw API 撰写的程序我们通常也称为SDK程序。也有人把Windows API称为SDK API。Borland 公司的C++ 编译器也支持相同的SDK API（那当然，因为Windows只有一套）。本书如果出现「SDK 程序」这样的名词，指的就是以 Windows raw API 完成的程序。

MFC**—Microsoft Foundation Classes** 的缩写，这是一个架构在Windows API之上的C++类库（C++ Class Library），意图使 Windows 程序设计过程更有效率，更符合面向对象的精神。MFC 在争取成为「Windows 类库标准」的路上声势浩大。Symantec C++以及WATCOM C/C++已向微软取得授权，在它的软件开发平台上供应 MFC。Borland C++也可以吃进MFC程序代码——啊，OWL 的地位益形尴尬了。

OWL**—Object Windows Library** 的缩写，这也是一个具备Application Framework 架势的C++ 类库，附含在 Borland C++ 之中。

Application Framework—在面向对象领域中，这是一个专有名词。关于它的意义，本书第５章有不少介绍。基本上它可以说是一个更有凝聚力，关联性更强的类库。并不是每一套C++类库都有资格称为Application Framework，不过MFC和OWL都可入列，IBM 的 Open Class Library 也是。Application Framework 当然不一定得是C++类库，Java和 Delphi 应该也都称得上。

为使全书文字流畅精简，我用了一些缩写字：

```
API - Application Programming Interface
DLL - Dynamic Link Library
GUI - Graphics User Interface
MDI - Multiple Document Interface
MFC - Microsoft Foundation Class
OLE - Object Linking & Embedded
OWL - Object Windows Library
SDK - Software Development Kit
SDI - Single Document Interface
UI  - User Interface
WinApp : Windows Application
```

以下是本书使用之中英文名词对照表：

```
Control          控制组件，如  Edit、ListBox、Button...。
drag & drop      拖放（鼠标左键按下，选中图示后拖动，然后放开）
Icon             图标（窗口缩小化后的小图样）
linked-list      串列
listbox          列表框、列表清单
notification     通告消息（发生于控制组件）
preemptive       强制性、先占式、优先权式
process          进程（一个执行起来的程序）
queue            队列
template        C++ 有所谓的class template，一般译为类模板；Windows 有所谓的dialog template，我
window class     窗口类（不是一种C++ 类）
window focus     窗口焦点（拥有焦点之窗口，将可以获得键盘输入）
class            类
object           对象
constructor      构造函数
destructor       析构函数
operator         运算符
override         改写
overloading      重载 ，亦有他书译为「过荷」
Encapsulation    封装
Inheritance      继承
Dynamic Binding  动态联编 ，亦即后期联编（late binding）
virtual function 虚函数
Polymorphism     多态 ，亦有他书译为「同名异式」
member function  成员函数
data member      成员变量，亦有他书译为「数据成员」
Base Class       基类，亦即父类
Derived Class    派生类，亦即子类
```

本书符号习惯

斜体字表示函数、常数、变量、语言保留字、宏、识别代码等等，例如：

```
CreateWindow            这是Win32 函数
Strtok                  这是C Runtime函数库的函数
WM_CREATE               这是Windows消息
ID_FILE_OPEN            这是资源识别代码（ID）
CDocument::Serialize    这是MFC类的成员函数
m_pNewViewClass         这是MFC类的成员变量
BEGIN_MESSAGE_MAP       这是MFC宏
Public                  这是C++语言保留字
```

第一篇 勿在浮砂築高台

第 1 章 Win32 基本程序观念

> 程序设计领域里，每一个人都想飞。
>
> 但是，还没学会走之前，连跑都别想！

虽然这是一本深入讲解MFC 程序设计的书，我仍坚持要安排这第一章，介绍 Win32 的 基本程序设计原理（也就是所谓的SDK 程序设计原理）。从来不曾学习过在「事件驱动（event driven）系统」中撰写「以消息为基础（message based）之应用程序」者，能否一步跨入MFC 领域，直接以application framework 开发 Windows 程序，我一直抱持怀疑的态度。虽然有了MFC（或任何其它的application framework），你可以继承一整组类，从而快速得到一个颇具规模的程序，但是 Windows 程序的运作 本质（Message Based，Event Driven）从来不曾也不会改变。如果你不能了解其髓，空有 其皮其肉或其骨，是不可能有所精进的，即使能够操控 wizard，充其量却也只是个 puppet，对于手上的程序代码，没有自主权。

我认为学习MFC之前，必要的基础是，对于Windows程序的事件驱动特性的了解（包括消息的产生、获得、分派、判断、处理），以及对 C++ 多态（polymorphism）的精确体会。本章所提出的，是我对第一项必要基础的探讨，你可以从中获得关于 Windows 程序的诞生与死亡，以及多任务环境下程序之间共存的观念。至于第二项基础，将由第二章为你夯实。

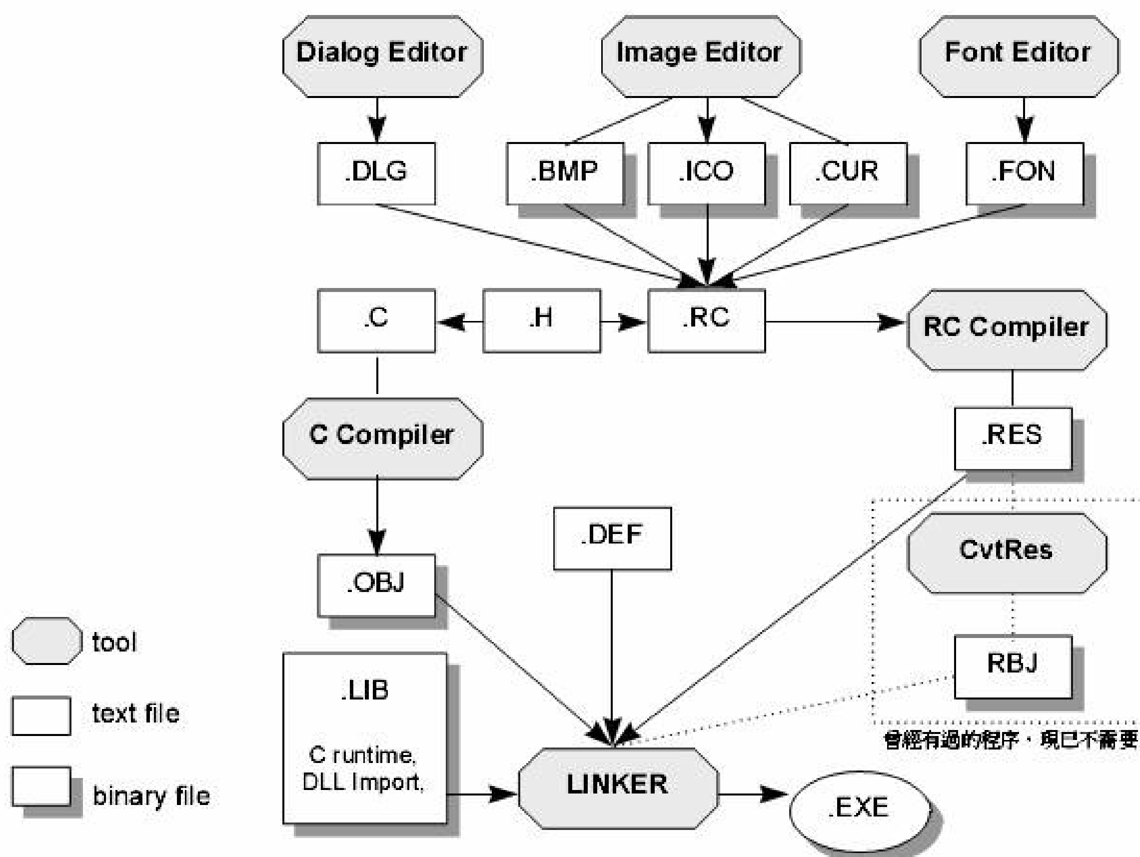


图1-1 一个32位Windows SDK程序的开发流程

需要什么函数库（.LIB）

众所周知Windows 支持动态链接。换句话说，应用程序所调用的Windows API 函数是在「运行时」才链接上的。那么，「链接时期」所需的函数库做什么用？有哪些？

并不是延伸文件名为 .dll 者才是动态链接函数库（DLL，Dynamic Link Library），事实上 .exe、.dll、.fon、.mod、.drv、.ocx 都是所谓的动态链接函数库。

Windows 程序调用的函数可分为 C Runtimes 以及 Windows API 两大部分。早期的 C Runtimes 并不支持动态链接，但Visual C++ 4.0 之后已支持，并且在32 位操作系统中已不再有small/medium/large 等内存模式之分。以下是它们的命名规则与使用时机：

- LIBC.LIB - 这是C Runtime 函数库的静态链接版本。
- MSVCRT.LIB - 这是C Runtime 函数库动态链接版本（MSVCRT40.DLL）的 import 函数库。如果链接此一函数库，你的程序执行时必须有 MSVCRT40.DLL 在场。

另一组函数，Windows API，由操作系统本身（主要是 Windows 三大模块 GDI32.DLL 和 USER32.DLL 和 KERNEL32.DLL）提供（注）。虽说动态链接是在运行时才发生「链接」事实，但在链接时期，链接器仍需先为调用者（应用程序本身）准备一些适当的资讯，才能够在运行时顺利「跳」到 DLL 执行。如果该 API 所属之函数库尚未加载，系统也才因此知道要先行加载该函数库。这些适当的信息放在所谓的「import 函数库」中。32 位 Windows 的三大模块所对应的 import 函数库分别为 GDI32.LIB 和 USER32.LIB 和 KERNEL32.LIB。

Windows 发展至今，逐渐加上的一些新的 API 函数（例如 Common Dialog、ToolHelp） 并不放在GDI和USER 和KERNEL三大模块中，而是放在诸如COMMDLG.DLL、TOOLHELP.DLL之中。如果要使用这些APIs，链接时还得加上这些DLLs 所对应的import函数库，诸如COMDLG32.LIB 和 TH32.LIB。

很快地，在稍后的范例程序“Generic” 的makefile中，你就可以清楚看到链接时期所需的各式各样函数库（以及各种链接器选项）。

需要什么头文件（.H）

所有Windows程序都必须含入WINDOWS.H。早期这是一个巨大的头文件，大约有5000 行左右，Visual C++ 4.0 已把它切割为各个较小的文件，但还以 WINDOWS.H 总括之。 除非你十分清楚什么 API 动作需要什么头文件，否则为求便利，单单一个 WINDOWS.H 也就是了。

不过，WINDOWS.H 只照顾三大模块所提供的API 函数，如果你用到其它 system DLLs，例如 COMMDLG.DLL 或 MAPI.DLL 或 TAPI.DLL 等等，就得含入对应的头文件，例如 COMMDLG.H 或 MAPI.H 或 TAPI.H 等等。

以消息为基础，以事件驱动之（message based, event driven）

Windows 程序的进行系依靠外部发生的事件来驱动。换句话说，程序不断等待（利用一个 while 循环），等待任何可能的输入，然后做判断，然后再做适当的处理。上述的「输入」是由操作系统捕捉到之后，以消息形式（一种数据结构）进入程序之中。操作系统如何捕捉外围设备（如键盘和鼠标）所发生的事件呢？噢，USER 模块掌管各个外围的驱动程序，它们各有侦测循环。

如果把应用程序获得的各种「输入」分类，可以分为由硬件装置所产生的消息（如鼠标移动或键盘被按下），放在系统队列（system queue）中，以及由 Windows 系统或其它Windows 程序传送过来的消息，放在程序队列（application queue）中。以应用程序的眼光来看，消息就是消息，来自哪里或放在哪里其实并没有太大区别，反正程序调用GetMessage API就取得一个消息，程序的生命靠它来推动。所有的GUI系统，包括UNIX的X Window 以及OS/2的 Presentation Manager，都像这样，是以消息为基础的事件驱动系统。

可想而知，每一个Windows程序都应该有一个循环如下：

```
MSG msg;
    while (GetMessage(&msg, NULL, NULL, NULL)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
} // 以上出现的函数都是Windows API 函数
消息，也就是上面出现的MSG结构，其实是Windows内定的一种数据格式:
/* Queued message structure */
typedef struct tagMSG
{
    HWND      hwnd;
    UINT      message;  // WM_xxx，例如 WM_MOUSEMOVE，WM_SIZE...
    WPARAM    wParam;
    LPARAM    lParam;
    DWORD     time;
    POINT     pt;
} MSG;
```

接受并处理消息的主角就是窗口。每一个窗口都应该有一个函数负责处理消息，程序员必须负责设计这个所谓的「窗口函数」（window procedure，或称为window function）。如果窗口获得一个消息，这个窗口函数必须判断消息的类，决定处理的方式。

以上就是Windows程序设计最重要的观念。至于窗口的产生与显示，十分简单，有专门的API 函数负责。稍后我们就会看到 Windows 程序如何把这消息的取得、分派、处理动作表现出来。

一个具体而微的Win32程序

许多相关书籍或文章尝试以各种方式简化Windows程序的第一步，因为单单一个Hello程序就要上百行，怕把大家吓坏了。我却宁愿各位早一点接触正统写法，早一点看到全 貌。Windows 的东西又多又杂，早一点一窥全貌是很有必要的。而且你会发现，经过有 条理的解释之后，程序代码的多寡其实构不成什么威胁（否则无字天书最适合程序员阅 读）。再说，上百进程序代码哪算得了什么！

你可以从图1-2 得窥 Win32 应用程序的本体与操作系统之间的关系。Win32 程序中最具代表意义的动作已经在该图显示出来，完整的程序代码展示于后。本章后续讨论都围绕着此一程序。

稍后会出现一个makefile。关于makefile的语法，可能已经不再为大家所熟悉了。我想我有必要做个说明。所谓makefile，就是让你能够设定某个文件和某个文件相比——比较其产生日期。由其比较结果来决定要不要做某些你所指定的动作。例如：

```
generic.res : generic.rc generic.h
rc generic.rc
```

意思就是拿冒号（:）左边的generic.res和冒号右边的generic.rc 和generic.h 的文件日期相比。只要右边任一文件比左边的文件更新，就执行下一行所指定的动作。这动作可以是任何命令列动作，本例为rc generic.rc。

因此，我们就可以把不同文件间的依存关系做一个整理，以makefile 语法描述，以产生必要的编译、链接动作。makefile 必须以NMAKE.EXE（Microsoft 工具）或MAKE.EXE（Borland 工具）处理之，或其它编译器套件所附的同等工具（可能也叫做MAKE.EXE）处理之。

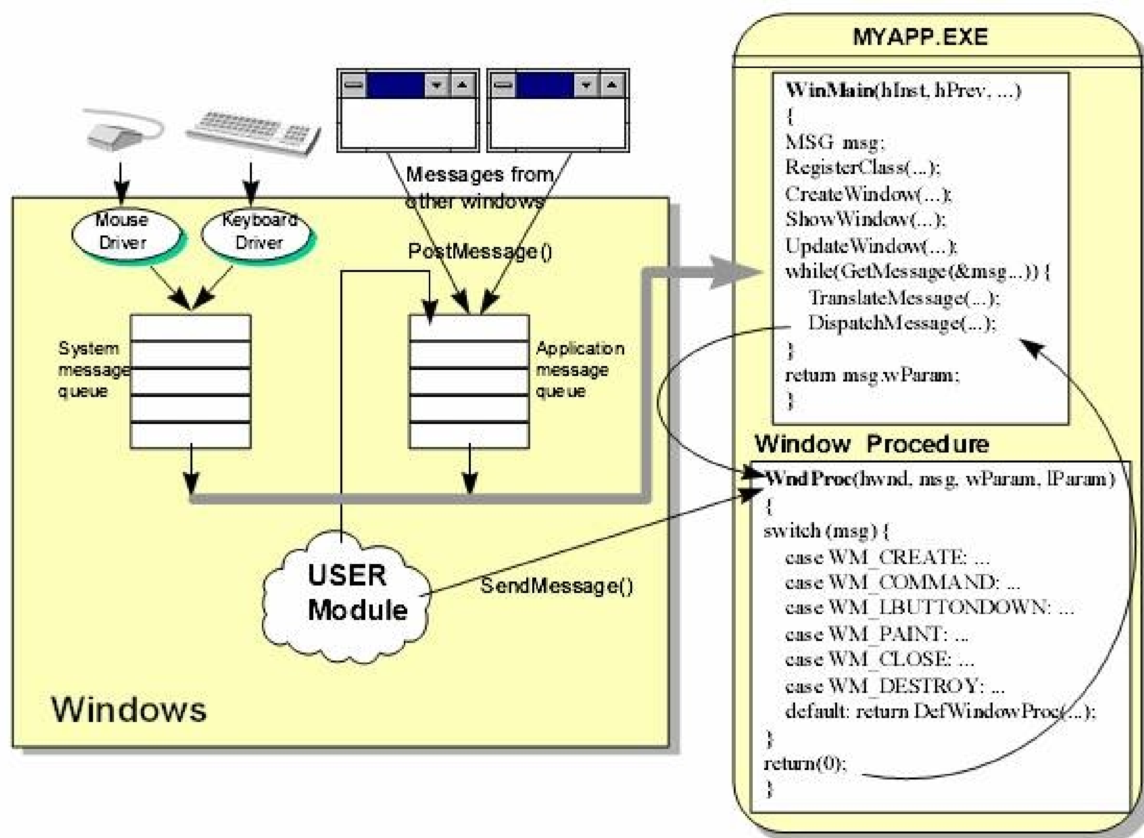


图 1-2 Windows 程序的本体与操作系统之间的关系

Generic.mak（请在DOS窗口中执行nmake generic.mak。环境设定请参考 p.224）

```
#0001  # filename : generic.mak
#0002  # make file for generic.exe (Generic Windows Application)
#0003  # usage : nmake generic.mak (Microsoft C/C++ 9.00) (Visual C++ 2.x)
#0004  # usage : nmake generic.mak (Microsoft C/C++ 10.00) (Visual C++ 4.0)
#0005
#0006  all: generic.exe
#0007
#0008  generic.res : generic.rc generic.h
#0009      rc generic.rc
#0010
#0011  generic.obj : generic.c generic.h
#0012      cl-c-W3-Gz-D_X86_-DWIN32 generic.c
#0013
#0014  generic.exe : generic.obj generic.res
#0015      link/MACHINE:I386 -subsystem:windows generic.res generic.obj \
#0016          libc.lib kernel32.lib user32.lib gdi32.lib
```

Generic.h

```
#0001  //---------------------------------------------------------------
#0002  // 文件名 : generic.h
#0003  //---------------------------------------------------------------
#0004  BOOL InitApplication(HANDLE);
#0005  BOOL InitInstance(HANDLE, int);
#0006  LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
#0007  LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);
```

Generic.c（粗体代表Windows API函数或宏）

```
#0001  //---------------------------------------------------------------
#0002  //             Generic - Win32 程序的基础写法
#0003  //                Top Studio * J.J.Hou
#0004  // 文件名     : generic.c
#0005  // 作者       : 侯俊杰
#0006  // 编译链接   : 请参考 generic.mak
#0007  //---------------------------------------------------------------
#0008
#0009  #include <windows.h>   // 每一个 Windows 程序都需要含入此文件
#0010  #include "resource.h"  // 内含各个 resource IDs
#0011  #include "generic.h" // 本程序之含入文件
#0012
#0013  HINSTANCE _hInst; // Instance handle
#0014  HWND      _hWnd;
#0015
#0016  char _szAppName[] = "Generic"; // 程序名称
#0017  char _szTitle[]= "Generic Sample Application"; // 窗口标题
#0017  char _szTitle[]
#0018
#0019  //---------------------------------------------------------------
#0020  // WinMain - 程序进入点
#0021  //---------------------------------------------------------------
#0022  int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
#0023                      LPSTR lpCmdLine, int nCmdShow)
#0024  {
#0025    MSG msg;
#0026
#0027    UNREFERENCED_PARAMETER(lpCmdLine); // 避免编译时的警告
#0028
#0029    if (!hPrevInstance)
#0030        if (!InitApplication(hInstance))
```

```
#0031           return (FALSE);
#0032
#0033   if (!InitInstance(hInstance, nCmdShow))
#0034       return (FALSE);
#0035
#0036   while (GetMessage(&msg, NULL, 0, 0)) {
#0037       TranslateMessage(&msg);
#0038       DispatchMessage(&msg);
#0039   }
#0040
#0041   return (msg.wParam); // 传回 PostQuitMessage 的参数
#0042 }
#0043 //---------------------------------------------------------------
#0044 // InitApplication - 注册窗口类
#0045 //---------------------------------------------------------------
#0046 BOOL InitApplication(HINSTANCE hInstance)
#0047 {
#0048  WNDCLASS  wc;
#0049
#0050  wc.style = CS_HREDRAW | CS_VREDRAW;
#0051  wc.lpfnWndProc= (WNDPROC)WndProc; // 窗口函数
#0052  wc.cbClsExtra= 0;
#0053  wc.cbWndExtra= 0;
#0054  wc.hInstance= hInstance;
#0055  wc.hIcon= LoadIcon(hInstance, "jjhouricon");
#0056  wc.hCursor= LoadCursor(NULL, IDC_ARROW);
#0057  wc.hbrBackground = GetStockObject(WHITE_BRUSH);  // 窗口后台颜色
#0058  wc.lpszMenuName  = "GenericMenu"; // .RC 所定义的窗体
#0059  wc.lpszClassName = _szAppName;
#0060
#0061  return (RegisterClass(&wc));
#0062 }
#0063 //---------------------------------------------------------------
#0064 // InitInstance - 产生窗口
#0065 //---------------------------------------------------------------
#0066 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
#0067 {
#0068    _hInst = hInstance; // 储存为全局变量，方便使用。
#0069
#0070    _hWnd = CreateWindow(
#0071                           _szAppName,
#0072                           _szTitle,
#0073                           WS_OVERLAPPEDWINDOW,
#0074                           CW_USEDEFAULT,
#0075                           CW_USEDEFAULT,
#0076                           CW_USEDEFAULT,
#0077                           CW_USEDEFAULT,
#0078                           NULL,
#0079                           NULL,
#0080                           hInstance,
#0081                           NULL
#0082                           );
#0083
#0084  if (!_hWnd)
#0085      return (FALSE)
#0086
#0087  ShowWindow(_hWnd, nCmdShow); // 显示窗口
#0088  UpdateWindow(_hWnd); // 送出 WM_PAINT
#0089  return (TRUE);
#0090 }
#0091 //---------------------------------------------------------------
#0092 // WndProc - 窗口函数
#0093 //---------------------------------------------------------------
#0094 LRESULT CALLBACK WndProc(HWND hWnd,     UINT message,
#0095                          WPARAM wParam, LPARAM lParam)
#0096 {
#0097   int wmId, wmEvent;
#0098
#0099   switch (message) {
#0100   case WM_COMMAND:
#0101
#0102       wmId    = LOWORD(wParam);
```

```
#0103         wmEvent = HIWORD(wParam);
#0104
#0105         switch (wmId) {
#0106         case IDM_ABOUT:
#0107             DialogBox(_hInst,
#0108                       "AboutBox",// 对话框资源名称
#0109                       hWnd, // 父窗口
#0110                       (DLGPROC)About  // 对话框函数名称
#0111                       );
#0112             break;
#0113
#0114         case IDM_EXIT:    // 使用者想结束程序。处理方式与 WM_CLOSE 相同。
#0116             DestroyWindow(hWnd);
#0117             break;
#0118
#0119         default:
#0120             return (DefWindowProc(hWnd, message, wParam, lParam));
#0121         }
#0122         break;
#0123
#0124      case WM_DESTROY:  // 窗口已经被摧毁 (程序即将结束)。
#0125          PostQuitMessage(0);
#0126          break;
#0127
#0128      default:
#0129          return (DefWindowProc(hWnd, message, wParam, lParam));
#0130     }
#0131     return (0);
#0132   }
#0133   //----------------------------------------------------------------
#0134   // About - 对话框函数
#0135   //----------------------------------------------------------------
#0136   LRESULT CALLBACK About(HWND hDlg,     UINT message,
#0137                          WPARAM wParam, LPARAM lParam)
#0138   {
#0139      UNREFERENCED_PARAMETER(lParam); // 避免编译时的警告
#0140
#0141      switch (message) {
#0142          case WM_INITDIALOG:
#0143              return (TRUE); // TRUE 表示我已处理过这个消息
#0144
#0145          case WM_COMMAND:
#0146              if (LOWORD(wParam) == IDOK
#0147                  ||LOWORD(wParam) == IDCANCEL) {
#0148                  EndDialog(hDlg, TRUE);
#0149                  return (TRUE);  // TRUE 表示我已处理过这个消息
#0150              }
#0151              break;
#0152   }
#0153      return (FALSE); // FALSE 表示我没有处理这个消息
#0154   }
```

Generic.rc

```
#0001  //---------------------------------------------------------------
#0002  // 文件名 : generic.rc
#0003  //---------------------------------------------------------------
#0004  #include "windows.h"
#0005  #include "resource.h"
#0006
#0007  jjhouricon ICON DISCARDABLE "jjhour.ico"
#0008
#0009  GenericMenu MENU DISCARDABLE
#0010  BEGIN
#0011         POPUP "&File"
#0012         BEGIN
#0013             MENUITEM "&New",             IDM_NEW, GRAYED
#0014             MENUITEM "&Open...",         IDM_OPEN, GRAYED
#0015             MENUITEM "&Save",            IDM_SAVE, GRAYED
#0016             MENUITEM "Save &As...",      IDM_SAVEAS, GRAYED
#0017             MENUITEM SEPARATOR
#0018             MENUITEM "&Print...",        IDM_PRINT, GRAYED
#0019             MENUITEM "P&rint Setup...",  IDM_PRINTSETUP, GRAYED
#0020             MENUITEM SEPARATOR
#0021             MENUITEM "E&xit",            IDM_EXIT
#0022         END
#0023         POPUP "&Edit"
#0024         BEGIN
#0025             MENUITEM "&Undo\tCtrl+Z",    IDM_UNDO, GRAYED
#0026             MENUITEM SEPARATOR
#0027             MENUITEM "Cu&t\tCtrl+X",     IDM_CUT, GRAYED
#0028             MENUITEM "&Copy\tCtrl+C",    IDM_COPY, GRAYED
#0029             MENUITEM "&Paste\tCtrl+V",   IDM_PASTE, GRAYED
#0030             MENUITEM "Paste &Link",      IDM_LINK, GRAYED
#0031             MENUITEM SEPARATOR
#0032             MENUITEM "Lin&ks...",        IDM_LINKS, GRAYED
#0033         END
#0034         POPUP "&Help"
#0035         BEGIN
#0036             MENUITEM "&Contents",               IDM_HELPCONTENTS, GRAYED
#0037             MENUITEM "&Search for Help On...",  IDM_HELPSEARCH, GRAYED
#0038             MENUITEM "&How to Use Help",        IDM_HELPHELP, GRAYED
#0039             MENUITEM SEPARATOR
#0040             MENUITEM "&About Generic...",       IDM_ABOUT
#0041         END
#0042   END
#0043
#0044   AboutBox DIALOG DISCARDABLE  22, 17, 144, 75
#0045   STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
#0046   CAPTION "About Generic"
#0047   BEGIN
#0048       CTEXT "Windows 95", -1,0, 5,144,8
#0049       CTEXT "Generic Application",-1,0,14,144,8
#0050       CTEXT "Version 1.0",-1,0,34,144,8
#0051       DEFPUSHBUTTON "OK", IDOK,53,59,32,14,WS_GROUP
#0052   END
```

程序进入点 WinMain

main 是一般C程序的进入点：

```
int main(int argc, char *argv[ ], char *envp[ ]);
{
    ...
}
```

WinMain 则是Windows 程序的进入点：

```
int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine,     int nCmdShow)
{
...
} // 在 Win32 中 CALLBACK 被定义为 __stdcall，是一种函数调用习惯，关系到参数挤压到堆栈的次序，以及处
```

当Windows的「外壳」（shell，例如 Windows 3.1的程序管理员或Windows 95的文件总管）侦测到使用者意欲执行一个 Windows 程序，于是调用加载器把该程序加载，然后调用C startup code，后者再调用WinMain，开始执进程序。WinMain 的四个参数由作业系统传递进来。

窗口类之注册与窗口之诞生

一开始，Windows 程序必须做些初始化工作，为的是产生应用程序的工作舞台：窗口。这没有什么困难，因为API函数**CreateWindow**完全包办了整个巨大的工程。但是窗口产生之前，其属性必须先设定好。所谓属性包括窗口的「外貌」和「行为」，一个窗口的边框、颜色、标题、位置等等就是其外貌，而窗口接收消息后的反应就是其行为（具体地说就是指窗口函数本身）。程序必须在产生窗口之前先利用 API 函数 RegisterClass设定属性（我们称此动作为注册窗口类）。RegisterClass 需要一个大型数据结构WNDCLASS 做为参数，CreateWindow 则另需要11个参数。

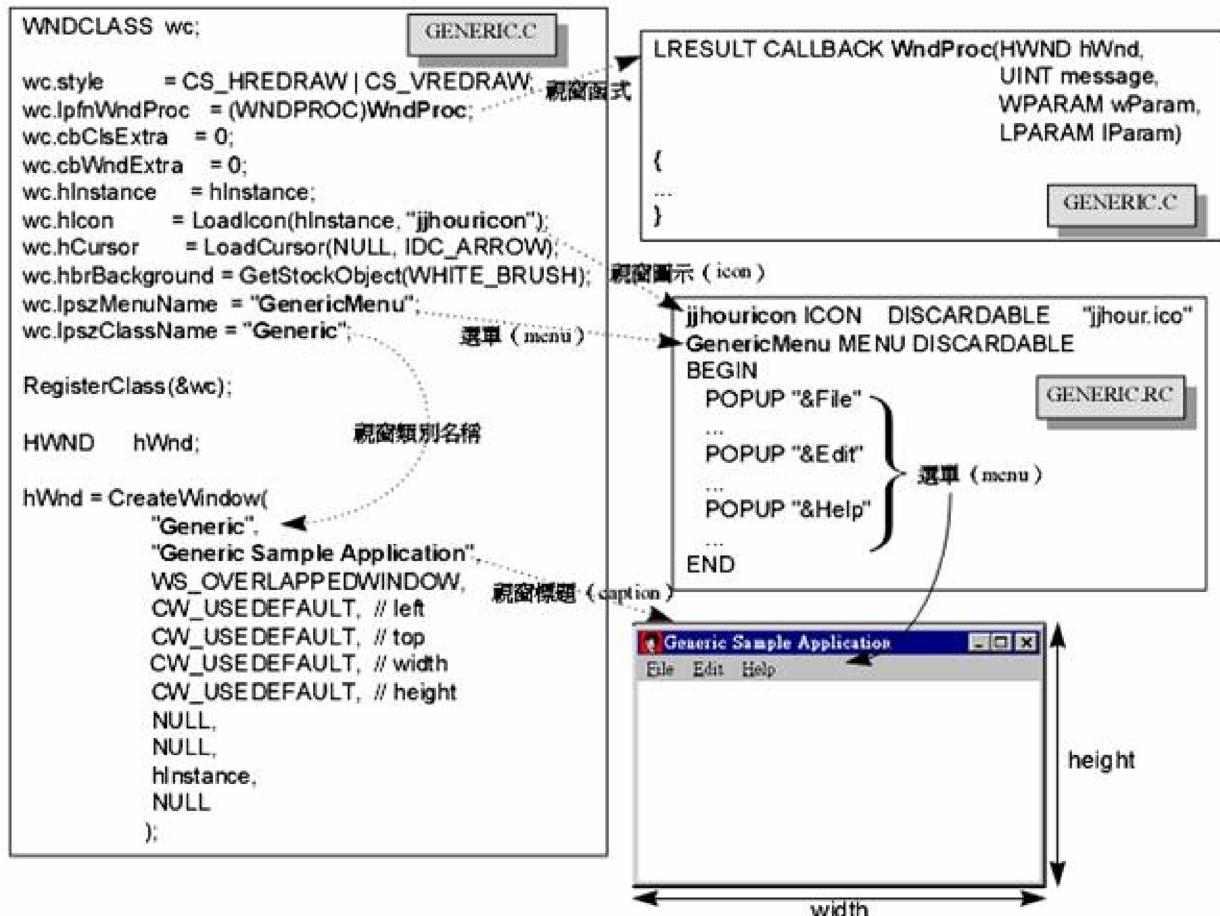


图1-3 RegisterClass与CreateWindow

从图1-3可以清楚看出一个窗口类牵扯的范围多么广泛，其中**wc.lpfnWndProc**所指定的函数就是窗口的行为中枢，也就是所谓的窗口函数。注意，**CreateWindow**只产生窗口，并不显示窗口，所以稍后我们必须再利用 **ShowWindow** 将之显示在屏幕上。又，我们希望先传送个WM_PAINT给窗口，以驱动窗口的绘图动作，所以调用**UpdateWindow**。消息传递的观念暂且不表，稍后再提。

请注意，在Generic程序中，RegisterClass 被我包装在InitApplication 函数之中，CreateWindow 则被我包装在InitInstance函数之中。这种安排虽非强制，却很普遍：

```
int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine,     int nCmdShow)
{
    if (!hPrevInstance)
    if (!InitApplication(hInstance))
    return (FALSE);
    if (!InitInstance(hInstance, nCmdShow))
    return (FALSE);
    ...
}
BOOL InitApplication(HINSTANCE hInstance)
{
    WNDCLASS  wc;
    ...
    return (RegisterClass(&wc));
}
//--------------------------------------------------
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    _hWnd = CreateWindow(...);
    ...
}
```

两个函数（InitApplication 和 InitInstance）的名称别具意义：

- 在Windows 3.x时代，窗口类只需注册一次，即可供同一程序的后续每一个执行个体（instance）使用（之所以能够如此，是因为所有进程共在一个地址空间中），所以我们把RegisterClass这个动作安排在“只有第一个执行个体才会进入” 的InitApplication 函数中。至于此一进程是否是某个程序的第一个执行个体，可由WinMain的参数hPrevInstance 判断之；其值由系统传入。
- 产生窗口，是每一个执行个体（instance）都得做的动作，所以我们把CreateWindow 这个动作安排在「任何执行个体都会进入」的InitInstance 函数中。

以上情况在 Windows NT 和 Windows 95 中略有变化。由于 Win32 程序的每一个执行个体（instance）有自己的地址空间，共享同一窗口类已不可能。但是由于 Win32 系统令hPrevInstance永远为0，所以我们仍然得以把RegisterClass和CreateWindow按旧习惯安排。既符合了新环境的要求，又兼顾到了旧原始代码的兼容。

InitApplication 和InitInstance 只不过是两个自定函数，为什么我要对此振振有词呢？原因是MFC 把这两个函数包装成CWinApp 的两个虚成员函数。第 6 章「MFC 程序的生与死」对此有详细解释。

消息循环

初始化工作完成后，WinMain 进入所谓的消息循环：

```
while (GetMessage(&msg,...)) {
    TranslateMessage(&msg);  // 转换键盘消息
    DispatchMessage(&msg);   // 分派消息
}
```

其中的**TranslateMessage**是为了将键盘消息转化，**DispatchMessage** 会将消息传给窗口函数去处理。没有指定函数名称，却可以将消息传送过去，岂不是很玄？这是因为消息发生之时，操作系统已根据当时状态，为它标明了所属窗口，而窗口所属之窗口类又已经明白标示了窗口函数（也就是 wc.lpfnWndProc 所指定的函数)，所以 DispatchMessage自有脉络可寻。请注意图 1-2 所示，DispatchMessage 经过USER模块的协助，才把消息交到窗口函数手中。

消息循环中的GetMessage是Windows 3.x非强制性（non-preemptive）多任务的关键。应用程序藉由此动作，提供了释放控制权的机会：如果消息队列上没有属于我的消息，我就把机会让给别人。通过程序之间彼此协调让步的方式，达到多任务能力。Windows 95 和Windows NT 具备强制性（preemptive）多任务能力，不再非靠 GetMessage释放CPU 控制权不可，但程序写法依然不变，因为应用程序仍然需要靠消息推动。它还是需要抓消息！

窗口的生命中枢：窗口函数

消息循环中的DispatchMessage把消息分配到哪里呢？它通过USER模块的协助，送到该窗口的窗口函数去了。窗口函数通常利用switch/case方式判断消息种类，以决定处置方式。由于它是被Windows系统所调用的（我们并没有在应用程序任何地方调用此函数），所以这是一种call back函数，意思是指「在你的程序中，被 Windows 系统调用」的函数。这些函数虽然由你设计，但是永远不会也不该被你调用，它们是为 Windows 系统准备的。

程序进行过程中，消息由输入装置，经由消息循环的抓取，源源传送给窗口并进而送到窗口函数去。窗口函数的体积可能很庞大，也可能很精简，依该窗口感兴趣的消息数量多寡而定。至于窗口函数的型式，相当一致，必然是：

```
LRESULT CALLBACK WndProc(HWND hWnd,
UINT message,
WPARAM wParam,
LPARAM lParam)
```

注意，不论什么消息，都必须被处理，所以switch/case 指令中的default: 处必须调用DefWindowProc，这是 Windows 内部预设的消息处理函数。

窗口函数的 wParam 和 lParam 的意义，因消息之不同而异。wParam 在 16 位环境中是16位，在 32 位环境中是 32 位。因此，参数内容（格式）在不同作业环境中就有了变化。

我想很多人都会问这个问题：为什么 Windows Programming Modal 要把窗口函数设计为一个call back 函数？为什么不让程序在抓到消息（GetMessage）之后直接调用它就好了？原因是，除了你需要调用它，有很多时候操作系统也要调用你的窗口函数（例如当某个消息产生或某个事件发生）。窗口函数设计为 callback 形式，才能开放出一个接口给操作系统调用。

消息映射（**Message Map**）的雏形

有没有可能把窗口函数的内容设计得更模块化、更一般化些？下面是一种作法。请注意，以下作法是 MFC「消息映射表」（第 9 章）的雏形，我所采用的结构名称和变量名称， 都与MFC 相同，藉此让你先有个暖身。

首先，定义一个MSGMAP_ENTRY结构和一个dim宏：

```
struct MSGMAP_ENTRY {
    UINT nMessage;
    LONG (*pfn)(HWND, UINT, WPARAM, LPARAM);
};
#define dim(x) (sizeof(x) / sizeof(x[0]))
```

请注意 MSGMAP_ENTRY 的第二元素 pfn 是一个函数指针，我准备以此指针所指之函数处理nMessage 消息。这正是面向对象观念中把「数据」和「处理数据的方法」封装起来的一种具体实现，只不过我们用的不是C++语言。

接下来，组织两个数组_messageEntries[ ]和_commandEntries[ ]，把程序中欲处理的消息以及消息处理例程的关联性建立起来：

```
// 消息与处理例程之对照表
struct MSGMAP_ENTRY _messageEntries[] =
{
    WM_CREATE,          OnCreate,
    WM_PAINT,           OnPaint,
    WM_SIZE,            OnSize,
    WM_COMMAND,         OnCommand,
    WM_SETFOCUS,        OnSetFocus,
    WM_CLOSE,           OnClose,
    WM_DESTROY,         OnDestroy,
}；这是消息        这是消息处理例程
// Command-ID 与处理例程之对照表格
struct MSGMAP_ENTRY _commandEntries =
{
    IDM_ABOUT,                OnAbout,
    IDM_FILEOPEN,             OnFileOpen,
    IDM_SAVEAS,               OnSaveAs,
};这是WM_COMMAND命令项    这是命令处理例程
```

于是窗口函数可以这么设计：

```
// 窗口函数
LRESULT CALLBACK WndProc(HWND hWnd,     UINT message,
WPARAM wParam, LPARAM lParam)
{
    int i;
    for(i=0; i < dim(_messageEntries); i++) {  // 消息对照表
    if (message == _messageEntries[i].nMessage)
    return((*_messageEntries[i].pfn)(hWnd,message,wParam,lParam));
    }
    return(DefWindowProc(hWnd, message, wParam, lParam));
}
// OnCommand——专门处理WM_COMMAND
LONG OnCommand(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int i;
    for(i=0; i < dim(_commandEntries); i++) {  // 命令项目对照表
    if (LOWORD(wParam) == _commandEntries[i].nMessage)
    return((*_commandEntries[i].pfn)(hWnd, message, wParam, lParam));
    }
    return(DefWindowProc(hWnd, message, wParam, lParam));
}
//----------------------------------------------------------------------
LONG OnCreate(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    ...
}
//----------------------------------------------------------------------
LONG OnAbout(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    ...
}
```

这么一来，WndProc和OnCommand永远不必改变，每有新要处理的消息，只要在 _messageEntries[ ] 和 _commandEntries[ ] 两个数组中加上新元素，并针对新消息撰写新的处理例程即可。

这种观念以及作法就是MFC的Message Map的雏形。MFC把其中的动作包装得更好更精致（当然因此也就更复杂得多），成为一张庞大的消息地图；程序一旦获得消息，就可以按图上溯，直到被处理为止。我将在第3章简单模拟MFC的Message Map，并在第9章「消息映射与循环」中详细探索其完整内容。

对话框的运作

Windows 的对话框依其与父窗口的关系，分为两类：

1.「令其父窗口除能，直到对话框结束」，这种称为modal 对话框。

2.「父窗口与对话框共同运行」，这种称为modeless 对话框。

比较常用的是 modal 对话框。我就以Generic 的About 对话框做为说明范例。为了做出一个对话框，程序员必须准备两样东西：

1. 对话框模板（dialog template）。这是在 RC 文件中定义的一个对话框外貌，以各种方式决定对话框的大小、字形、内部有哪些控制组件、各在什么位置...等等。

2. 对话框函数（dialog procedure）。其类型非常类似窗口函数，但是它通常只处理WM_INITDIALOG和WM_COMMAND两个消息。对话框中的各个控制组件也都是小小窗口，各有自己的窗口函数，它们以消息与其管理者（父窗口，也就是对话框）沟通。而所有的控制组件传来的消息都是 WM_COMMAND，再由其参数分辨哪一种控制组件以及哪一种通告（notification）。

Modal 对话框的启动与结束，靠的是DialogBox 和EndDialog 两个API 函数。请看图 1-4。

对话框处理过消息之后，应该传回TRUE；如果未处理消息，则应该传回FALSE。这是因为你的对话框函数之上层还有一个系统提供的预设对话框函数。如果你传回FALSE，该预设对话框函数就会接手处理。

模块定义文件（.DEF）

Windows 程序需要一个模块定义文件，将模块名称、程序节区和数据节区的内存特性、 模块堆积（heap）大小、堆栈（stack）大小、所有 callback 函数名称...等等登记下来。下面是个实例：

```
NAME              Generic
DESCRIPTION      ‘Generic Sample'
EXETYPE           WINDOWS
```

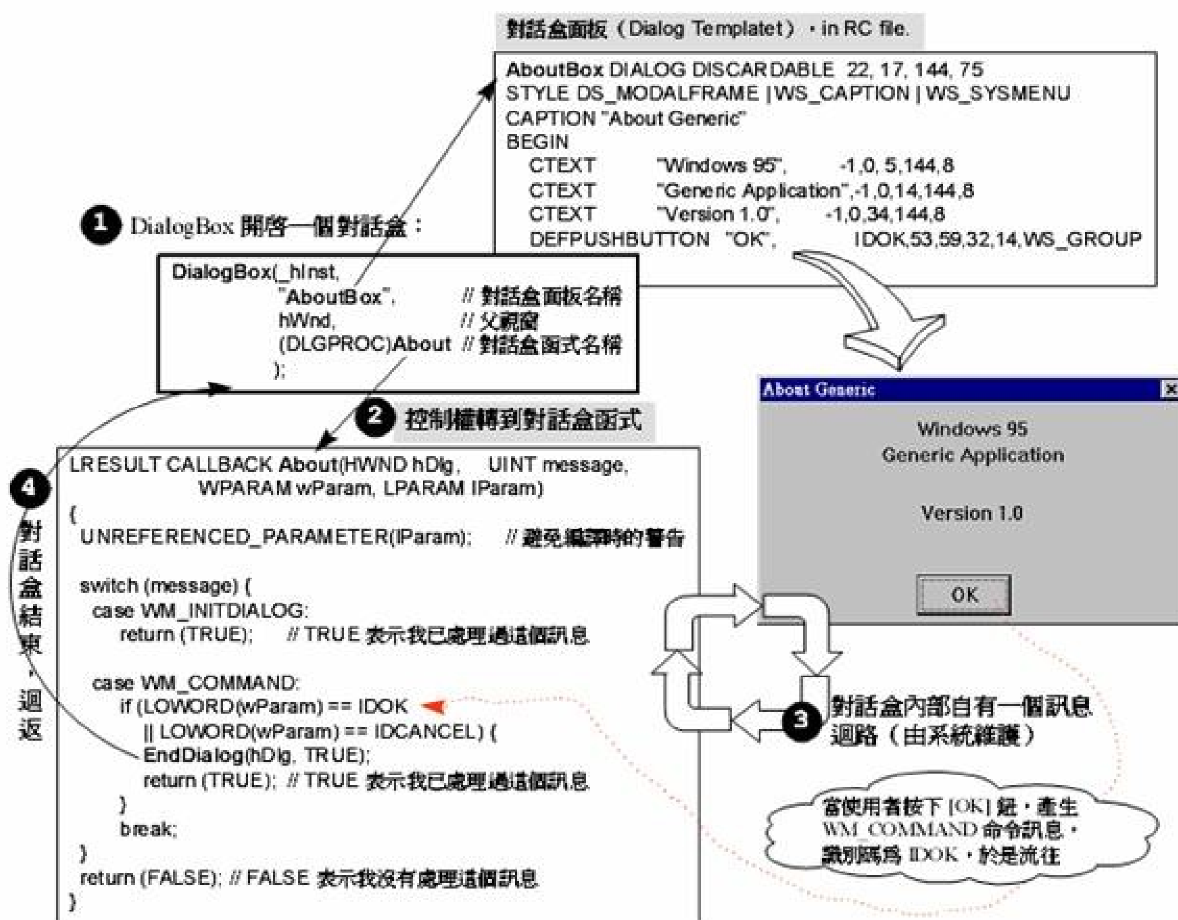


图1-4 对话框的诞生、运作、结束

```
STUB            'WINSTUB.EXE'
CODE              PRELOAD DISCARDABLE
DATA              PRELOAD MOVEABLE MULTIPLE
HEAPSIZE          4096
STACKSIZE         10240
EXPORTS
MainWndProc @1
AboutBox    @2
```

在Visual C++整合环境中开发程序，不再需要特别准备.DEF文件，因为模块定义文件中的设定都有默认值。模块定义文件中的 STUB 指令用来指定所谓的 stub 程序（埋在 Windows程序中的一个 DOS 程序，你所看到的 This Program Requires Microsoft Windows 或 This Program Can Not Run in DOS mode 就是此程序发出来的），Win16 允许程序员自设一个 stub 程序，但 Win32 不允许，换句话说在 Win32 之中Stub 指令已经失效。

资源描述文件（.RC）

RC文件是一个以文字描述资源的地方。常用的资源有九项之多，分别是 ICON、CURSOR、BITMAP、FONT、DIALOG、MENU、ACCELERATOR、STRING、VERSIONINFO。还可能有新的资源不断加入，例如Visual C++ 4.0就多了一种名为TOOLBAR的资源。这些文字描

述需经过RC编译器，才产生可使用的二进位代码。本例Generic示范ICON、MENU和DIALOG三种资源。

Windows 程序的生与死

我想你已经了解Windows程序的架构以及它与Windows系统之间的关系。对Windows 消息种类以及发生时机的透彻了解，正是程序设计的关键。现在我以窗口的诞生和死亡，说明消息的发生与传递，以及应用程序的兴起与结束，请看图 1-5 及图 1-6。

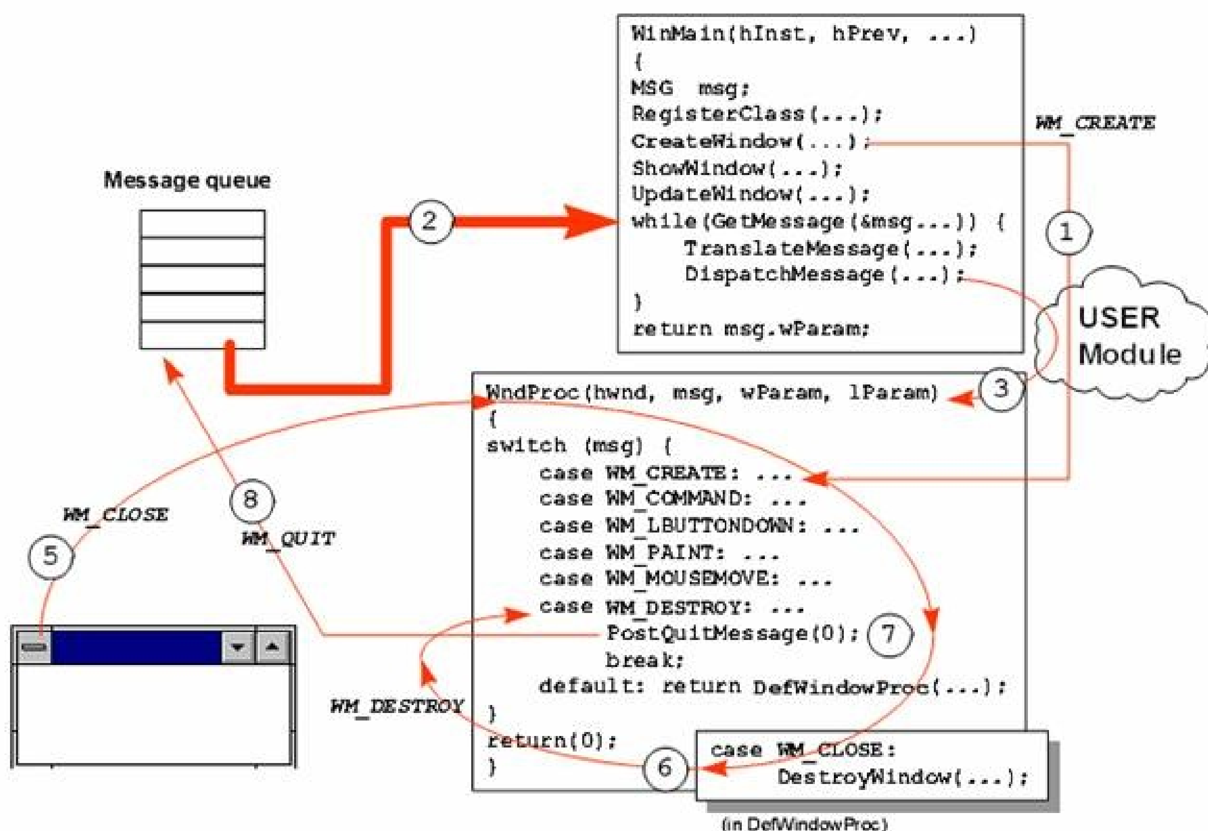


图1-5 窗口的生命周期（详细说明请看图1-6）

1. 程序初始化过程中调用CreateWindow，为程序建立了一个窗口，做为程序的屏幕舞台。CreateWindow 产生窗口之后会送出WM_CREATE直接给窗口函数，后者于是可以在此时机做些初始化动作（例如配置内存、开文件、读初始数据...）。

2. 程序活着的过程中，不断以GetMessage从消息贮列中抓取消息。如果这个消息是WM_QUIT，GetMessage 会传回0而结束while循环，进而结束整个程序。

3. DispatchMessage 通过Windows USER 模块的协助与监督，把消息分派至窗口函数。消息将在该处被判别并处理。

4. 程序不断进行2.和3.的动作。

5. 当使用者按下系统菜单中的Close命令项，系统送出WM_CLOSE。通常程序的窗口函数不拦截此消息，于是DefWindowProc 处理它。

6. DefWindowProc收到WM_CLOSE后， 调用DestroyWindow 把窗口清除 。DestroyWindow 本身又会送出WM_DESTROY。

7. 程序对WM_DESTROY的标准反应是调用PostQuitMessage。

8. PostQuitMessage 没什么其它动作，就只送出WM_QUIT 消息，准备让消息循环中的GetMessage 取得，如步骤2，结束消息循环。

图1-6 窗口的生命周期（请对照图1-5）

为什么结束一个程序复杂如斯？因为操作系统与应用程序职责不同，二者是互相合作的关系，所以必需各做各的份内事，并互以消息通知对方。如果不依据这个游戏规则，可能就会有麻烦产生。你可以作一个小实验，在窗口函数中拦截 WM_DESTROY，但不调用PostQuitMessage。你会发现当选择系统菜单中的 Close 时，屏幕上这个窗口消失了，（因为窗口摧毁及数据结构的释放是 DefWindowProc 调用 DestroyWindow 完成的），但是应用程序本身并没有结束（因为消息循环结束不了），它还留存在内存中。

闲置时间的处理：**OnIdle**

所谓闲置时间（idle time），是指「系统中没有任何消息等待处理」的时间。举个例子，没有任何程序使用定时器（timer，它会定时送来 WM_TIMER），使用者也没有碰触键盘和鼠标或任何外围，那么，系统就处于所谓的闲置时间。

闲置时间常常发生。不要认为你移动鼠标时产生一大堆的 WM_MOUSEMOVE，事实上夹杂在每一个WM_MOUSEMOVE之间就可能存在许多闲置时间。毕竟，计算机速度超乎想像。

后台工作最适宜在闲置时间完成。传统的 SDK 程序如果要处理闲置时间，可以以下列循环取代 WinMain 中传统的消息循环：

```
while (TRUE) {
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) {
        if (msg.message == WM_QUIT)
        break;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else {
        OnIdle();
    }
}
```

原因是PeekMessage和GetMessage的性质不同。它们都是到消息队列中抓消息，如果抓不到，程序的主线程（primary thread，是一个 UI 线程）会被操作系统虚悬住。当操作系统再次回来照顾此一线程，而发现消息队列中仍然是空的，这时候两个API函数的行为就有不同了：

- GetMessage 会过门不入，于是操作系统再去照顾其它人。
- PeekMessage 会取回控制权，使程序得以执行一段时间。于是上述消息循环进入OnIdle函数中。

第 6 章的HelloMFC将示范如何在MFC程序中处理所谓的idle time。

Console 程序

说到Windows 程序，一定得有WinMain、消息循环、窗口函数。即使你只产生一个对话窗（Dialog Box）或消息窗（Message Box），也有隐藏在 Windows API（DialogBox 和 MessageBox）内里的消息循环和窗口函数。

过去那种单单纯纯纯的C/C++程序，有着简单的main和printf的好时光到哪里去了？夏天在阴凉的树荫下嬉戏，冬天在温暖的炉火边看书，啊，Where did the good times go？

其实说到Win32 程序，并不是每个都如 Windows GUI 程序那么复杂可怖。是的，你可以在 Visual C++ 中写一个"DOS-like" 程序，而且仍然可以调用部分的、不牵扯到图形用户接口（GUI）的Win32 API。这种程序称为console 程序。甚至你还可以在console程序中使用部分的MFC 类（同样必须是与GUI 没有关连的），例如处理数组、串列等数据结构的collection classes（CArray、 CList、 CMap）、与文件有关的CFile、CStdioFile。

我在BBS论坛上看到很多程序设计初学者，还没有学习C/C++，就想直接学习Visual C++。并不是他们好高骛远，而是他们以为Visual C++是一种特殊的C++语言。吃过苦头的过来人以为初学所说的 Visual C++ programming 是指MFC programming，所以大吃一惊（没有一点C++基础就要学习MFC programming，当然是大吃一惊）。

在Visual C++中写纯种的C/C++程序？当然可以！不牵扯任何窗口、对话窗、控制元件，那就是console程序啰。虽然我这本书没有打算照顾 C++ 初学者，然而我还是决定把console 程序设计的一些相关心得放上来，同时也是因为我打算以 console 程序完成稍后的多线程程序范例。第 3 章的MFC六大技术仿真程序也都是console 程序。

其实，除了"DOS-like"，console 程序还另有妙用。如果你的程序和使用者之间是以巨量文字来互动，或许你会选择使用edit 控制组件（或MFC 的CEditView）。但是你知道，计算机在一个纯粹的「文字窗口」（也就是console 窗口）中处理文字的显现与卷动比较快，你的程序动作也比较简单。所以，你也可以在 Windows 程序中产生console 窗口，独立出来作业。

这也许不是你所认知的console 程序。总之，有这种混合式的东西存在。

这一节将以我自己的一个极简易的个人备份软件JBACKUP为实例，说明Win32 console 程序的撰写，以及如何在其中使用 Win32 API（其实直接调用就是了）。再以另一个极小的程序MFCCON示范MFC console程序（用到了MFC的CStudioFile和CString）。对于这么小的程序而言，实在不需动用到整合环境下的什么项目管理。至于复杂一点的程序，就请参考第 4 章最后一节「Console 程序的项目管理」。

Console 程序与 DOS 程序的差别

不少人把DOS程序和console程序混为一谈，这是不对的。以下是各方面的比较。

制造方式

在Windows环境下的DOS Box中，或是在Windows版本的各种C++编译器套件的整合环境（IDE）中（第 4 章「Console 程序项目管理」），利用 Windows 编译器、链接器做出来的程序，都是所谓Win32 程序。如果程序是以main为进入点，调用C runtime函数和「不牵扯GUI」的Win32 API函数，那么就是一个console 程序，console窗口将成为其标准输入和输出装置（cin 和cout）。

过去在DOS环境下开发的程序，称为DOS程序，它也是以main为程序进入点，可以调用C runtime函数。但，当然，不可能调用Win32 API 函数。

程序能力

过去的DOS程序仍然可以在Windows的DOS Box中跑（Win95 的兼容性极高，WinNT 的兼容性稍差）。

Console 程序当然更没有问题。由于console程序可以调用部分的Win32 API（尤其是KERNEL32.DLL 模块所提供的那一部分），所以它可以使用Windows提供的各种高阶功能。它可以产生进程（processes），产生线程（threads）、取得虚拟内存的信息、刺探操作系统的各种数据。但是它不能够有华丽的外表——因为它不能够调用与GUI有关的各种API 函数。

DOS程序和console程序两者都可以做printf输出和cout输出，也都可以做scanf 输入和cin 输入。

可执行文件格式

DOS 程序是所谓的MZ格式（MZ 是Mark Zbikowski的缩写，他是DOS系统的一位主要构造者）。Console 程序的格式则和所有的Win32 程序一样，是所谓的PE（Portable Executable）格式，意思是它可以被拿到任何Win32 平台上执行。

Visual C++ 附有一个DUMPBIN工具软件，可以观察PE文件格式。拿它来观察本节的JBACKUP 程序和MFCCON 程序（以及第 3 章的所有程序），得到这样的结果：

```
H:\u004\prog\jbackup.01>dumpbin /summary jbackup.exe
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997\. All rights reserved.
Dump of file jbackup.exe
File Type: EXECUTABLE IMAGE
Summary
5000 .data
1000 .idata
1000 .rdata
5000 .text
拿它来观察DOS程序，则得到这样的结果：
C:\UTILITY>dumpbin /summary dsize.exe
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997\. All rights reserved.
Dump of file dsize.exe
DUMPBIN : warning LNK4094: "dsize.exe" is an MS-DOS executable;
use EXEHDR to dump it
Summary
```

Console 程序的编译链接

你可以写一个makefile，编译时指定常数 /D_CONSOLE，链接时指定subsystem 为 console，如下：

```
#0001  # filename : pedump.mak
#0002  # make file for pedump.exe
#0003  # usage : nmake pedump.msc (Visual C++ 5.0)
#0004
#0005  all : pedump.exe
#0006
#0007  pedump.exe: pedump.obj exedump.obj objdump.obj common.obj
#0008      link /subsystem:console /incremental:yes \
#0009           /machine:i386 /out:pedump.exe \
#0010           pedump.obj common.obj exedump.obj objdump.obj \
#0011           kernel32.lib user32.lib
#0012
#0013  pedump.obj : pedump.c
#0014      cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c pedump.c
#0015
#0016  common.obj : common.c
#0017      cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c common.c
#0018
#0019  exedump.obj : exedump.c
#0020      cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c exedump.c
#0021
#0022  objdump.obj : objdump.c
#0023      cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c objdump.c
```

如果是很简单的情况，例如本节的JBACKUP只有一个C 原始代码，那么这样也行（在命令列之下）：

```
cl jbackup.c <ENTER>            ←将获得 jbackup.exe
```

注意，环境变量要先设定好（请参考本章稍早的「如何产生 Generic.exe」一节）。

第 3 章的Frame_ 程序则是这样完成的：

```
cl my.cpp mfc.cpp <ENTER>       ←将获得 my.exe
```

至于到底该链接哪些链接库，全让CL.EXE去伤脑筋就好了。

JBACKUP ：Win32 Console 程序设计

撰写console 程序，有几个重点请注意：

1. 进入点为main。

2. 可以使用printf、scanf、cin、cout 等标准输出入装置。

3. 可以调用和GUI 无关的Win32 API。

我的这个JBACKUP 程序可以有一个或两个参数，用法如下：

```
C:\SomeoneDir>JBACKUP SrcDir [DstDir]
```

例如 JBACKUP g: k:

将磁盘目录SrcDir中的新文件复制到磁盘目录DstDir，并将DstDir的赘余文件删除。

如果没有指定DstDir，预设为k:（那是我的可写入光驱-- MO-- 的代码啦）

并将k: 的磁盘目录设定与SrcDir相同。

例如JBACKUP g: ，而目前g: 是g:\u002\doc，那么相当于把g:\u002\doc备份到k:\u002\doc中，并删除k:\u002\doc的赘余文件。

JBACK检查SrcDir中所有的文件和DstDir中所有的文件，把比较新的文件从SrcDir中复制到DstDir去，并把DstDir中多出来的文件删除，使ScrDir和DstDir的文件保持完全相同。之所以不做 xcopy 完全复制动作，为的是节省复制时间（做为备份装置，通常是软盘或磁带或可擦写光盘 MO，读写速度并不快）。

JBACKUP 没有能力处理SrcDir 底下的子目录文件。如果要处理子目录，漂亮的作法是使用递归（recursive），但是有点伤脑筋，这一部分留给你了。我的打字速度还算快，多切换几次磁盘目录不是问题，呵呵呵。

JBACKUP使用以下数个Win32 APIs：

- GetCurrentDirectory
- FindFirstFile
- FindNextFile
- CompareFileTime
- CopyFile

- DeleteFile

在处理完毕命令列参数中的SrcDir和DstDir后，JBACKUP先把SrcDir的所有文件（不含子目录文件）搜寻一遍，储存在一个数组srcFiles[ ]中，每个数组元素是一个我自定的 SRCFILE 数据结构：

```
typedef struct _SRCFILE
    {
    WIN32_FIND_DATA fd;
    BOOL bIsNew;
} SRCFILE;
SRCFILE  srcFiles[FILEMAX];
WIN32_FIND_DATA fd;
// prepare srcFiles[]...
bRet = TRUE;
iSrcFiles = 0;
hFile = FindFirstFile(SrcDir, &fd);
while (hFile != INVALID_HANDLE_VALUE && bRet)
{
    if (fd.dwFileAttributes == FILE_ATTRIBUTE_ARCHIVE) {
        srcFiles[iSrcFiles].fd = fd;
        srcFiles[iSrcFiles].bIsNew = FALSE;
        iSrcFiles++;
    }
    bRet = FindNextFile(hFile, &fd);
}
```

再把DstDir中的所有文件（不含子目录文件）搜寻一遍，储存在一个destFiles[ ] 数组中，每个数组元素是一个我自定的 DESTFILE 数据结构：

```
typedef struct _DESTFILE
{
    WIN32_FIND_DATA fd;
    BOOL bMatch;
} DESTFILE;
DESTFILE destFiles[FILEMAX];
WIN32_FIND_DATA fd;
bRet = TRUE;
iDestFiles = 0;
hFile = FindFirstFile(DstDir, &fd);
while (hFile != INVALID_HANDLE_VALUE && bRet)
{
    if (fd.dwFileAttributes == FILE_ATTRIBUTE_ARCHIVE) {
        destFiles[iDestFiles].fd = fd;
        destFiles[iDestFiles].bMatch = FALSE;
        iDestFiles++;
    }
    bRet = FindNextFile(hFile, &fd);
}
```

然后比对srcFiles[ ]和destFiles[ ]之中的所有文件名称以及建文件日期，找出 scrFiles[ ]中的哪些文件比desFiles[ ]中的文件更新，然后将其bIsNew字段设为 TRUE。同时也对存在于desFiles[ ]中而不存在于srcFiles[ ]中的文件，令其bMatch字段为 FALSE。

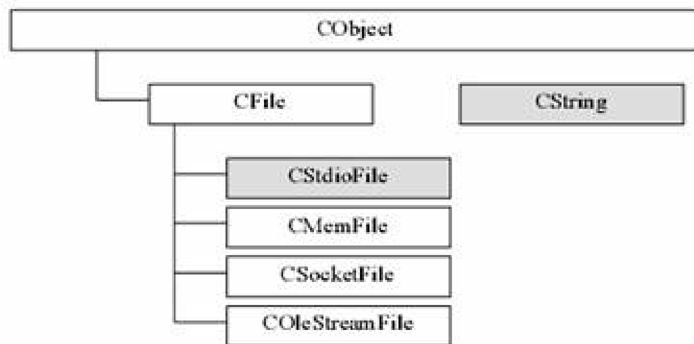
最后，检查srcFiles[ ]中的所有文件，将bIsNew字段为TRUE者，复制到DstDir 去。并检查destFiles[ ] 中的所有文件，将bMatch字段为FALSE者统统删除。

MFCCON ：MFC Console 程序设计

当你的进度还在第 1 章的Win32基本程序观念，我却开始讲如何设计一个MFC console程序，是否有点时地不宜？

是有一点！所以我挑一个最单纯而无与别人攀缠纠葛的MFC类，写一个40 行的小程序。目标纯粹是为了做一个导入，并与 Win32 console 程序做一比较。

我所挑选的两个单纯的MFC类是CStdioFile和CString：



在MFC之中，CFile 用来处理正常的文件I/O 动作。CStdioFile 派生自CFile，一个CStdioFile对象代表以C runtime函数fopen所开启的一个stream文件。Stream文件有缓冲区，可以文字模式（预设情况）或二进位模式开启。

CString 对象代表一个字符串，是一个完全独立的类。

我的例子用来计算小于100的所有费伯纳契数列（Fabonacci sequence）。费伯纳契数列的计算方式是：

1. 头两个数为 1。

2. 接下来的每一个数是前两个数的和。

以下便是MFCCON.CPP 内容

```
#0005  // Build  : cl /MT mfccon.cpp  (/MT means Multithreading)
#0006
#0007  #include <afx.h>
#0008  #include <stdio.h>
#0010  int main()
#0011  {
#0012   int lo, hi;
#0013   CString str;
#0014   CStdioFile fFibo;
#0016       fFibo.OpenFIBO.DAT", CFile::modeWrite |
#0017                   CFile::modeCreate | CFile::typeText);
#0019       str.Formats\n", "Fibonacci sequencee, less than 100 :");
#0020       printf("%s", (LPCTSTR) str);
#0021       fFibo.WriteStringstr);
#0023       lo = hi = 1;
#0025       str.Format("%d\n", lo);
#0026       printf("%s", (LPCTSTR) str);
#0027       fFibo.WriteString(str);
#0029       while (hi < 100)
#0030       {
#0031            str.Format("%d\n", hi);
#0032            printf("%s", (LPCTSTR) str);
#0033            fFibo.WriteString(str);
#0034            hi = lo + hi;
#0035            lo = hi - lo;
#0036        }
#0038        fFibo.Close();
#0039        return 0;
#0040    }
```

以下是执行结果（在console 窗口和FIBO.DAT 文件中，结果都一样）：

```
Fibonacci sequencee, less than 100 : 1\n 1\n 2\n 3\n 5\n 8\n 13\n 21\n 34\n 55\n 89
```

这么简单的例子中，我们看到MFC Console 程序的几个重点：

1. 程序进入点仍为main

2. 需含入所使用之类的头文件（本例为AFX.H）

3. 可直接使用与GUI无关的MFC类（本例为CStdioFile和CString）

4. 编辑时需指定/MT，表示使用多线程版本的C runtime 函数库。

第 4 点需要多做说明。在MFC console 程序中一定要指定多线程版的C runtime 函数库，所以必须使用/MT 选项。如果不做这项设定，会出现这样的链接错误：

```
Microsoft (R) 32-Bit Incremental Linker Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997\. All rights reserved.
/out:mfccon.exe
mfccon.obj
nafxcw.lib(thrdcore.obj):error LNK2001:unresolved external symbol __endthreadex
nafxcw.lib(thrdcore.obj):error LNK2001:unresolved external symbol __beginthreadex
mfccon.exe : fatal error LNK1120: 2 unresolved externals
```

表示它找不到**beginthreadex**和endthreadex。怪了，我们的程序有调用它们吗？没有，但是MFC 有！这两个函数将在稍后与线程有关的小节中讨论。

什么是 C Runtime 函数库的多线程版本

当C runtime 函数库于1970s 年代产生出来时，PC 的内存容量还很小，多任务是个新奇观念，更别提什么多线程了。因此以当时产品为基础所演化的C runtime 函数库在多线程（multithreaded）的表现上有严重问题，无法被多线程程序使用。

利用各种同步机制（synchronous mechanism）如critical section、mutex、semaphore、event，可以重新开发一套支持多线程的runtime 函数库。问题是，加上这样的能力，可能导至程序代码大小和执行效率都遭受不良波及——即使你只启动了一个线程。

Visual C++ 的折衷方案是提供两种版本的C runtime 函数库。一种版本给单线程程序使用，一种版本给多线程程序使用。多线程版本的重大改变是，第一，变量如errno者现在变成每个线程各拥有一个。第二，多线程版中的数据结构以同步机制加以保护。

Visual C++ 一共有六个C runtime 函数库产品供你选择：

```
Single-Threaded（static）      libc.lib      898,826
Multithreaded（static）        libcmt.lib    951,142
Multithreaded DLL               msvcrt.lib    5,510,000
Debug Single-Threaded（static）libcd.lib     2,374,542
Debug Multithreaded（static）  libcmtd.lib   2,949,190
Debug Multithreaded DLL         msvcrtd.lib   803,418
```

Visual C++ 编译器提供下列选项，让我们决定使用哪一个C runtime函数库：

```
/ML    Single-Threaded（static）
/MT    Multithreaded（static）
/MD    Multithreaded DLL（dynamic import library）
/MLd   Debug Single-Threaded（static）
/MTd   Debug Multithreaded（static）
/MDd   Debug Multithreaded DLL（dynamic import library）
```

进程与线程（Process and Thread）

OS/2、Windows NT 以及 Windows 95 都支持多线程，这带给PC程序员一股令人兴奋的气氛。然而它带来的不全然是利多，如果不谨慎小心地处理线程的同步问题，程序的错误以及除错所花的时间可能使你发誓再也不碰「线程」这种东西。

我们习惯以进程（process）表示一个执行中的程序，并且以为它是CPU排程单位。事实上线程才是排程单位。

核心对象

首先让我解释什么叫作「核心对象」（kernel object）。「GDI对象」是大家比较熟悉的东西，我们利用GDI函数所产生的一支画笔（Pen）或一支画刷（Brush）都是所谓的「GDI对象」。但什么又是「核心对象」呢？

你可以说核心对象是系统的一种资源（噢，这说法对 GDI 对象也适用），系统对象一旦产生，任何应用程序都可以开启并使用该对象。系统给予核心对象一个计数值（usage count）做为管理之用。核心对象包括下列数种：

```
核心对象              产生方法
event                 CreateEvent
mutex                 CreateMutex
semaphore             CreateSemaphore
file                  CreateFile
file-mapping          CreateFileMapping
process               CreateProcess
thread                CreateThread
```

前三者用于线程的同步化：file-mapping 对象用于内存映射文件（memory mapping file），process和thread对象则是本节的主角。这些核心对象的产生方式（也就是我们所使用的API）不同，但都会获得一个handle做为识别；每被使用一次，其对应的计数值就加1。核心对象的结束方式相当一致，调用 CloseHandle 即可。

「process 对象」究竟做什么用呢？它并不如你想象中用来「执进程序代码」；不，程序代码的执行是线程的工作，「process 对象」只是一个数据结构，系统用它来管理进程。

一个进程的诞生与死亡

执行一个程序，必然就产生一个进程（process）。最直接的程序执行方式就是在shell（如Win95 的文件总管或Windows 3.x 的文件管理员）中以鼠标双击某一个可执行文件图示（假设其为 App.exe），执行起来的App 进程其实是shell调用CreateProcess启动的。

让我们看看整个流程：

1. shell 调用CreateProcess启动App.exe。

2. 系统产生一个「进程核心对象」，计数值为1。

3. 系统为此进程建立一个4GB 地址空间。

4. 加载器将必要的代码加载到上述地址空间中，包括App.exe 的程序、数据，以及所需的动态链接函数库（DLLs）。载入器如何知道要载入哪些DLLs呢？它们被记录在可执行文件（PE 文件格式）的 .idata section中。

5. 系统为此进程建立一个线程，称为主线程（primary thread）。线程才是CPU时间的分配对象。

6. 系统调用C runtime 函数库的Startup code。

7. Startup code调用App程序的WinMain函数。

8. App程序开始运作。

9. 使用者关闭App主窗口，使WinMain中的消息循环结束掉，于是WinMain 结束。

10. 回到Startup code。

11. 回到系统，系统调用ExitProcess结束进程。

可以说，通过这种方式执行起来的所有 Windows 程序，都是 shell 的子进程。本来，母进程与子进程之间可以有某些关系存在，但 shell 在调用 CreateProcess 时已经把母子之间的脐带关系剪断了，因此它们事实上是独立个体。稍后我会提到如何剪断子进程的脐带。

产生子进程

你可以写一个程序，专门用来启动其他的程序。关键就在于你会不会使用CreateProcess。这个API函数有众多参数：

```
CreateProcess(
    LPCSTR lpApplicationName,
    LPSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

第一个参数lpApplicationName指定可执行文件文件名。第二个参数lpCommandLine指定欲传给新进程的命令行（command line）参数。如果你指定了lpApplicationName，但没有扩展名，系统并不会主动为你加上 .EXE 扩展名；如果没有指定完整路径，系统就只在目前工作目录中寻找。但如果你指定lpApplicationName为NULL的话，系统会以lpCommandLine 的第一个「段落」（我的意思其实是术语中所谓的 token）做为可执行文件文件名；如果这个文件名没有指定扩展名，就采用预设的".EXE" 扩展名；如果没有指定路径，Windows 就依照五个搜寻路径来寻找可执行文件，分别是：

1. 调用者的可执行文件所在目录

2. 调用者的目前工作目录

3. Windows目录

4. Windows System目录

5. 环境变量中的path所设定的各目录

让我们看看实例：

```
CreateProcess("E:\\CWIN95\\NOTEPAD.EXE", "README.TXT",...);
```

系统将执行 E:\CWIN95\NOTEPAD.EXE，命令列参数是 "README.TXT"。如果我们这样子调用：

```
CreateProcess(NULL, "NOTEPAD README.TXT",...);
```

系统将依照搜寻次序，将第一个被找到的 NOTEPAD.EXE 执行起来，并转送命令列参数 "README.TXT" 给它。

建立新进程之前，系统必须做出两个核心对象，也就是「进程对象」和「线程对象」。CreateProcess 的第三个参数和第四个参数分别指定这两个核心对象的安全属性。至于第五个参数（TRUE 或FALSE）则用来设定这些安全属性是否要被继承。关于安全属性及其可被继承的性质，碍于本章的定位，我不打算在此介绍。

第六个参数dwCreationFlags可以是许多常数的组合，会影响到进程的建立过程。这些常数中比较常用的是CREATE_SUSPENDED，它会使得子进程产生之后，其主线程立刻被暂停执行。

第七个参数lpEnvironment可以指定进程所使用的环境变量区。通常我们会让子进程继承父进程的环境变量，那么这里要指定NULL。

第八个参数lpCurrentDirectory用来设定子进程的工作目录与工作磁盘。如果指定NULL，子进程就会使用父进程的工作目录与工作磁盘。

第九个参数lpStartupInfo是一个指向STARTUPINFO结构的指针。这是一个庞大的结构，可以用来设定窗口的标题、位置与大小，详情请看 API 使用手册。

最后一个参数是一个指向 PROCESS_INFORMATION 结构的指针：

```
typedef struct _PROCESS_INFORMATION {
HANDLE hProcess;
HANDLE hThread;
DWORD dwProcessId;
DWORD dwThreadId;
} PROCESS_INFORMATION;
```

当系统为我们产生「进程对象」和「线程对象」，它会把两个对象的handle 填入此结构的相关字段中，应用程序可以从这里获得这些handles。

如果一个进程想结束自己的生命，只要调用：VOID ExitProcess(UINT fuExitCode); 就可以了。如果进程想结束另一个进程的生命，可以使用：

```
BOOL TerminateProcess(HANDLE hProcess, UINT fuExitCode);
```

很显然，只要你有某个进程的handle，就可以结束它的生命。TerminateProcess 并不被建议使用，倒不是因为它的权力太大，而是因为一般进程结束时，系统会通知该进程所开启（所使用）的所有DLLs，但如果你以 TerminateProcess 结束一个进程，系统不会做这件事，而这恐怕不是你所希望的。

前面我曾说过所谓割断脐带这件事情，只要你把子进程以CloseHandle 关闭，就达到了目的。下面是个例子：

```
PROCESS-INFORMATION ProcInfo;
BOOL fSuccess;
fSuccess = CreateProcess(...,&ProcInfo);
if (fSuccess) {
    CloseHandle(ProcInfo.hThread);
    CloseHandle(ProcInfo.hProcess);
}
```

一个线程的诞生与死亡

程序代码的执行，是线程的工作。当一个进程建立起来，主线程也产生。所以每一个Windows 程序一开始就有了一个线程。我们可以调用CreateThread 产生额外的线程，系统会帮我们完成下列事情：

1. 配置「线程对象」，其handle将成为CreateThread的传回值。

2. 设定计数值为 1。

3. 配置线程的context。

4. 保留线程的堆栈。

5. 将context中的堆栈指针缓存器（SS）和指令指针缓存器（IP）设定妥当。

看看上面的态势，的确可以显示出线程是CPU分配时间的单位。所谓工作切换（context switch）其实就是对线程的context的切换。

程序若欲产生一个新线程，调用CreateThread 即可办到：

```
CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

第一个参数表示安全属性的设定以及继承，请参考API手册。Windows 95 忽略此一参数。第二个参数设定堆栈的大小。第三个参数设定「线程函数」名称，而该函数的参数则在这里的第四个参数设定。第五个参数如果是0，表示让线程立刻开始执行，如果是CREATE_SUSPENDED，则是要求线程暂停执行（那么我们 必须调用ResumeThread才能令其重新开始）。最后一个参数是个指向DWORD的指针，系统会把线程的ID放在这里。

上面我所说的「线程函数」是什么？让我们看个实例：

```
VOID   ReadTime(VOID);
HANDLE hThread;
DWORD  ThreadID;
hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ReadTime,
    NULL, 0, &ThreadID);
...
// thread 函数。不断利用GetSystemTime 取系统时间，并将结果显示在对话框_hWndDlg的IDE_TIMER
//字段上。
VOID ReadTime(VOID)
{
    char str[50];
    SYSTEMTIME st;
    while(1) {
        GetSystemTime(&st);
        sprintf(str,"%u:%u:%u", st.wHour, st.wMinute, st.wSecond);
        SetDlgItemText (_hWndDlg, IDE_TIMER, str);
        Sleep (1000);  // 延迟一秒。
    }
}
```

当CreateThread成功，系统为我们把一个线程该有的东西都准备好。线程的主体在哪里呢？就在所谓的线程函数。线程与线程之间，不必考虑控制权释放的问题，因为 Win32 操作系统是强制性多任务。

线程的结束有两种情况，一种是寿终正寝，一种是未得善终。前者是线程函数正常结束退出，那么线程也就自然而然终结了。这时候系统会调用 ExitThread 做些善后清理工作（其实线程中也可以自行调用此函数以结束自己）。但是像上面那个例子，线程根本是个无穷循环，如何终结？一者是进程结束（自然也就导至线程的结束），二者是别的线程强制以TerminateThread 将它终结掉。不过，TerminateThread 太过毒辣，非必要还是少用为妙（请参考API 手册）。

以 _beginthreadex 取代 CreateThread

别忘了Windows程序除了调用Win32 API，通常也很难避免调用任何一个C runtime函数。为了保证多线程情况下的安全，C runtime 函数库必须为每一个线程做一些记录工作。没有这些工作，C runtime 函数库就不知道要为每一个线程配置一块新的内存，做为线程的局部变量用。因此，CreateThread 有一个名为 _beginthreadex 的外包函数，负责额外的记录工作。

请注意函数名称的底线符号。它必须存在，因为这不是个标准的ANSI C runtime 函数。_beginthreadex 的参数和 CreateThread 的参数其实完全相同，不过其类型已经被「净化」了，不再有 Win32 类型包装。这原本是为了要让这个函数能够移植到其它操作系统，因为微软希望 _beginthreadex 能够被实现于其它平台，不需要和 Windows 有关、不需要含入 windows.h。但实际情况是，你还是得调用CloseHandle 以关闭线程，而CloseHandle 却是个 Win32 API，所以你还是需要含入windows.h、还是和Windows 脱离不了关系。微软空有一个好主意，却没能落实它。

把 _beginthreadex视为CreateThread的一个看起来比较有趣的版本，就对了：

```
unsigned long _beginthreadex(
    void *security,
    unsigned stack_size,
    unsigned(__stdcall *start_address)(void *),
    void *arglist,
    unsigned initflag,
    unsigned* thrdaddr
);
```

_beginthreadex 所传回的unsigned long事实上就是一个Win32 HANDLE，指向新线程。换句话说传回值和CreateThread相同，但 _beginthreadex 另外还设立了errno 和doserrno。

下面是一个最简单的使用范例：

```
#0001  #include <windows.h>
#0002  #include <process.h>
#0003  unsigned __stdcall myfunc(void* p);
#0005  void main()
#0006  {
#0007      unsigned long thd;
#0008      unsigned tid;
#0010      thd = _beginthreadex(NULL,
#0011                           0,
#0012                           myfunc,
#0013                           0,
#0014                           0,
#0015                           &tid );
#0016      if (thd != NULL)
#0017      {
#0018          CloseHandle(thd);
#0019       }
#0020  }
#0022  unsigned __stdcall myfuncs(void* p)
#0023  {
#0024  // do your job...
#0025  }
```

针对Win32 API ExitThread，也有一个对应的C runtime函数：_endthreadex。它只需要一个参数，就是由 _beginthreadex 第 6 个参数传回来的ID 值。

线程优先权（Priority）

优先权是排程的重要依据。优先权高的线程，永远先获得CPU的青睐。当然啦，作业系统会视情况调整各个线程的优先权。例如前台线程的优先权应该调高一些，后台线程的优先权应该调低一些。

线程的优先权范围从0（最低）到31（最高）。当你产生线程，并不是直接以数值指定其优先权，而是采用两个步骤。第一个步骤是指定「优先权等级（Priority Class）」给进程，第二步骤是指定「相对优先权」给该进程所拥有的线程。图 1-7 是优先权等级的描述，其中的代码在 CreateProcess 的 dwCreationFlags 参数中指定。如果你不指定，系统预设给的是 NORMAL_PRIORITY_CLASS——除非父进程是IDLE_PRIORITY_CLASS（那么子进程也会是IDLE_PRIORITY_CLASS）。Win32线程的优先权等级划分：

```
等级          代码                         优先权值
Idle          IDLE_PRIORITY_CLASS           4
Normal        NORMAL_PRIORITY_CLASS         9（前台）或 7（后台）
high          HIGH_PRIORITY_CLASS           13
realtime      REALTIME_PRIORITY_CLASS       24
```

- "idle" 等级只有在CPU 时间将被浪费掉时（也就是前一节所说的闲置时间）才执行。此等级最适合于系统监视软件，或屏幕保护软件。
- "normal" 是预设等级。系统可以动态改变优先权，但只限于 "normal" 等级。当进程变成前台，线程优先权提升为 9，当进程变成后台，优先权降低为7。
- "high" 等级是为了立即反应的需要，例如使用者按下Ctrl+Esc 时立刻把工作管理器（task manager）带出场。
- "realtime" 等级几乎不会被一般应用程序使用。就连系统中控制鼠标、键盘、磁盘状态重新扫描、Ctrl+Alt+Del 等的线程都比"realtime" 优先权还低。这种等级使用在「如果不在某个时间范围内被执行的话，数据就要遗失」的情况。这个等级一定得在正确评估之下使用之，如果你把这样的等级指定给一般的（并不会常常被阻塞的）线程，多任务环境恐怕会瘫痪，因为这个线程有如此高的优先权，其它线程再没有机会被执行。

上述四种等级，每一个等级又映射到某一范围的优先权值。IDLE 最低，*NORMAL* 次之，HIGH 又次之，*REALTIME* 最高。在每一个等级之中，你可以使用SetThreadPriority设定精确的优先权，并且可以稍高或稍低于该等级的正常值（范围是两个点数）。你可以把 SetThreadPriority 想象是一种微调动作。

```
SetThreadPriority 的参数                    微调幅度
THREAD_PRIORITY_LOWEST                        -2
THREAD_PRIORITY_BELOW_NORMAL                  -1
THREAD_PRIORITY_NORMAL                       不变
THREAD_PRIORITY_ABOVE_NORMAL                  +
THREAD_PRIORITY_HIGHEST                       +2
```

除以上五种微调，另外还可以指定两种微调常数：

```
SetThreadPriority 的参数    面对任何等级面对    "realtime" 等级
的调整结果：         的调整结果：
THREAD_PRIORITY_IDLE              1                    16
THREAD_PRIORITY_TIME_CRITICAL     15                   31
```

这些情况可以以图 1-8 作为总结。

```
优先权等级    idle  lowest   below   normal  above   highest   time
Normal           normal            critical
idle        1     2        3       4       5        6        15
normal(后台)  1     5        6       7       8        9        15
normal(前台)  1     7        8       9       10       11       15
high        1     11       12      13      14       15       15
realtime      16    22       23      24      25       26       31
```

图1-8 Win32线程优先权

多线程程序设计实例

我设计了一个MltiThrd 程序，一开始产生五个线程，优先权分别微调-2、-1、0、+1、+2，并且虚悬不执行：

```
HANDLE _hThread[5];  // global variable
...
LONG APIENTRY MainWndProc (HWND hWnd, UINT message, UINT wParam, LONG lParam)
{
    DWORD  ThreadID[5];
    static DWORD  ThreadArg[5] = {HIGHEST_THREAD,    // 0x00
    ABOVE_AVE_THREAD,  // 0x3F
    NORMAL_THREAD,     // 0x7F
    BELOW_AVE_THREAD,  // 0xBF
    LOWEST_THREAD      // 0xFF
    };    // 用来调整四方形颜色
    ...
    for(i=0; i<5; i++) // 产生 5 个 threads
        _hThread[i] = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)ThreadProc,
        &ThreadArg[i],
        CREATE_SUSPENDED
        &ThreadID[i]);
    // 设定 thread priorities
    SetThreadPriority(_hThread[0], THREAD_PRIORITY_HIGHEST);
    SetThreadPriority(_hThread[1], THREAD_PRIORITY_ABOVE_NORMAL);
    SetThreadPriority(_hThread[2], THREAD_PRIORITY_NORMAL);
    SetThreadPriority(_hThread[3], THREAD_PRIORITY_BELOW_NORMAL);
    SetThreadPriority(_hThread[4], THREAD_PRIORITY_LOWEST);
    ...
}
```

当使用者按下【Resume Threads】菜单项目后，五个线程如猛虎出柙，同时冲出来。这五个线程使用同一个线程函数ThreadProc。我在ThreadProc中以不断的Rectangle动作表示线程的进行。所以我们可以从画面上观察线程的进度。我并且设计了两种延迟方式，以利观察。

第一种方式是在每一次循环之中使用 Sleep(10)，意思是先睡10个毫秒，之后再醒来；这段期间，CPU可以给别人使用。第二种方式是以空循环30000次做延迟；空循环期间CPU不能给别人使用（事实上CPU正忙碌于那30000次空转）。

```
UINT _uDelayType=NODELAY;  // global variable
...
VOID ThreadProc(DWORD *ThreadArg) {
    RECT rect; HDC  hDC;
    HANDLE hBrush, hOldBrush;
    DWORD dwThreadHits = 0;
    int   iThreadNo, i;
    ...
    do{
        dwThreadHits++; // 计数器
        // 画出四方形，代表 thread 的进行
        Rectangle(hDC, *(ThreadArg), rect.bottom-(dwThreadHits/10),
            *(ThreadArg)+0x40, rect.bottom);
        // 延迟...
        if (_uDelayType == SLEEPDELAY)
        Sleep(10);
        else if (_uDelayType == FORLOOPDELAY)
        for (i=0; i<30000; i++);
        else // _uDelayType == NODELAY)
        {   }
    } while (dwThreadHits < 1000); // 巡回1000次
    ...
}
```

图 1-9 是执行画面。注意，先选择延迟方式（"for loop delay" 或 "sleep delay"），再按下【Resume Thread】。如果你选择”for loop delay”（图 1-9a），你会看到线程0（优先权最高）几乎一路冲到底，然后才是线程 1（优先权次之），然后是线程2（优先权再次之）...。但如果你选择的”sleep delay”（图 1-9b），所有线程不分优先权高低，同时行动。关于线程的排程问题，我将在第14章做更多的讨论。

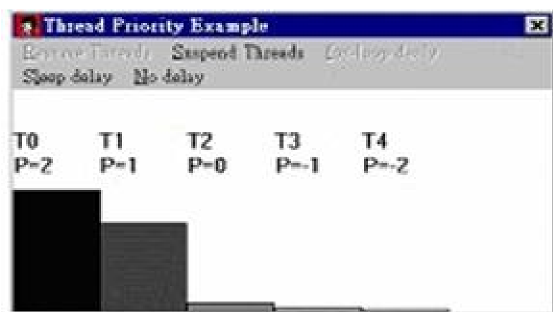


图1-9a MltiThrd.exe 的执行画面（”for loop delay”）

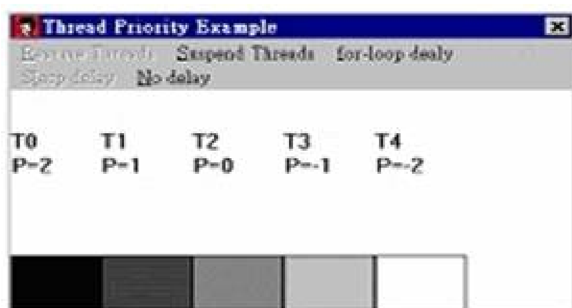


图1-9b MltiThrd.exe 的执行画面（”sleep delay”）

注意：为什么图 1-9a 中线程 1 尚未完成，线程 2~4 竟然也有机会偷得一点点 CPU 时间呢？这是排程器的巧妙设计，动态调整线程的优先权。是啊，总不能让优先权低的线程直到天荒地老，没有一点点获得。关于线程排程问题，第 14 章有更多的讨论。

第 2 章 C++ 的重要性质

> C++ 是一种扭转程序员思维模式的语言。
>
> 一个人思维模式的扭转，不可能轻而易举一蹴而成。

近来「面向对象」一词席卷了整个软件界。面向对象程序设计（ Object Oriented Programming）其实是一种观念，用什么语言实现它都可以。但，当然，面向对象程序语言（Object Oriented Programming Language）是专门为面向对象观念而发展出来的，以之完成面向对象的封装、继承、多态等特性自是最为便利。

C++ 是最重要的面向对象语言，因为它站在 C 语言的肩膀上，而 C 语言拥有绝对优势的使用者。C++ 并非纯粹的面向对象程序语言，不过有时候混血并不是坏事，纯种也不见得就多好。

所谓纯面向对象语言，是指不管什么东西，都应该存在于对象之中。JAVA 和 Small Talk都是纯面向对象语言。

如果你是C++的初学者，本章不适合你（事实上整本书都不适合你），你的当务之急是去买一本C++专书。一位专精Basic和Assembly语言的朋友问我，有没有可能不会C++而学会MFC？答案是当然没有可能。

如果你对C++一知半解，语法大约都懂了，语意大约都不懂，本章是我能够给你的最好礼物。我将从类与对象的关系开始，逐步解释封装、继承、多态、虚函数、动态联编。不只解释其操作方式，更要点出其意义与应用，也就是，为什么需要这些性质。

C++ 语言范围何其广大，这一章的主题挑选完全是以MFC Programming 所需技术为前提。下一章，我们就把这里学到的C++ 技术和OO观念应用到application framework 的仿真上，那是一个 DOS 程序，不牵扯 Windows。

类及其成员——谈封装（**encapsulation**）

让我们把世界看成是一个由对象（object）所组成的大环境。对象是什么？白一点说，「东西」是也！任何实际的物体你都可以说它是对象。为了描述对象，我们应该先把对象的属性描述出来。好，给「对象的属性」一个比较学术的名词，就是「类」（class）。

对象的属性有两大成员，一是数据，一是行为。在面向对象的术语中，前者常被称为property（Java 语言则称之为 field），后者常被称为 method。另有一双比较像程序设计领域的术语，名为 member variable（或 data member）和 member function。为求统一，本书使用第二组术语，也就是 member variable（成员变量）和 member function（成员函数）。一般而言，成员变量通常由成员函数处理之。

如果我以 CSquare 代表「四方形」这种类，四方形有 color，四方形可以 display。好，color 就是一种成员变量，display 就是一种成员函数：

```
CSquare square;      // 声明square 是一个四方形。
square.color = RED; // 设定成员变量。RED代表一个颜色值。
square.display();    // 调用成员函数。
```

下面是C++ 语言对于CSquare 的描述：

```
class CSquare  // 常常我们以 C 作为类名称的开头
{
private:
    int m_color; // 通常我们以 m_ 作为成员变量的名称开头
public:
    void display() { ... }
    void setcolor(int color) { m_color = color; }
};
```

成员变量可以只在类内被处理，也可以开放给外界处理。以数据封装的目的而言，自然是前者较为妥当，但有时候也不得不开放。为此，C++ 提供了private、public 和protected 三种修饰词。一般而言成员变量尽量声明为private，成员函数则通常声明为public。上例的m_color 既然声明为private，我们势必得准备一个成员函数setcolor，供外界设定颜色用。

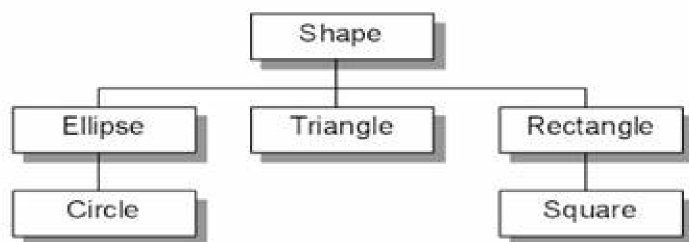
把数据声明为 private，不允许外界随意存取，只能通过特定的接口来操作，这就是面向对象的封装（encapsulation）特性。

基类与派生类：谈继承(Inheritance)

其它语言欲完成封装性质，并不太难。以 C 为例，在结构（struct）之中放置数据，以及处理数据的函数的指针（function pointer），就可得到某种程度的封装精神。

C++ 神秘而特有的性质其实在于继承。

矩形是形，椭圆形是形，三角形也是形。苍蝇是昆虫，蜜蜂是昆虫，蚂蚁也是昆虫。是的，人类习惯把相同的性质抽取出来，成立一个基类（base class），再从中衍化出派生类（derived class）。所以，关于形状，我们就有了这样的类阶层：



```
#0001  class CShape // 形状
#0002  {
#0003   private:
#0005       int m_color;
#0006   public:
#0007       void setcolor(int color) { m_color = color; }
#0008   };
#0010  class CRect : public CShape   // 矩形是一种形状
#0011  {                            //它会继承m_color 和setcolor()
#0012   public:
#0013       void display() { ... }
#0014   };
#0016  class CEllipse : public CShape // 椭圆形是一种形状
#0017  {                              //它会继承m_color 和setcolor()
#0018   public:
#0019       void display() { ... }
#0020   };
#0022  class CTriangle : public CShape // 三角形是一种形状
#0023  {                               //它会继承m_color 和setcolor()
#0024   public:
#0025       void display() { ... }
#0026   };
#0028  class CSquare : public CRect    // 四方形是一种矩形
#0029  {
#0030   public:
#0031       void display() { ... }
#0032   };
#0034  class CCircle : public CEllipse // 圆形是一种椭圆形
#0035  {
#0036   public:
#0037       void display() { ... }
#0038   };
```

于是你可以这么动作：

```
CSquare square;
CRect   rect1, rect2;
CCircle circle;
square.setcolor(1);   // 令 square.m_color = 1;
square.display();     // 调用 CSquare::display
rect1.setcolor(2);    // 于是 rect1.m_color = 2
rect1.display();      // 调用 CRect::display
rect2.setcolor(3);    // 于是 rect2.m_color = 3
rect2.display();      // 调用 CRect::display
circle.setcolor(4);   // 于是 circle.m_color = 4
circle.display();     // 调用 CCircle::display
```

注意以下这些事实与问题：

1. 所有类都由CShape 派生下来，所以它们都自然而然继承了CShape的成员，包括变量和函数。也就是说，所有的形状类都「暗自」具备了m_color 变量和setcolor函数。我所谓暗自（implicit），意思是无法从各派生类的声明中直接看出来。

2. 两个矩形对象rect1和rect2各有自己的m_color，但关于setcolor函数却是共享相同的CRect::setcolor（其实更应该说是CShape::setcolor）。我用这张图表示其间的关系：

```
对象 rect1            对象 rect2
m_color               m_color
↑                     ↑
this 指针             this 指针
```

```
CRect::setcolor(int color, CRect* this)
{
    this->m_color = color;
}//这个this参数是编译器自行为我们加上的，所以我说它是个 "隐藏指针"。
```

rect1.setcolor 和 rect2.setcolor 调用的都是 CRect::setcolor，后者之所以能分别处理不同对象的成员变量，完全是靠一个隐藏的this指针。

让我替你问一个问题：同一个函数如何处理不同的数据？为什么rect1.setcolor 和 rect2.setcolor 明明都是调用CRect::setcolor（其实也就是CShape::setcolor），却能够有条不紊地分别处理 rect1.m_color 和rect2.m_color？答案在于所谓的this 指针。下一节我就会提到它。

3. 既然所有类都有display动作，把它提升到老祖宗CShape去，然后再继承之，好吗？不好，因为display函数应该因不同的形状而动作不同。

4. 如果display不能提升到基类去，我们就不能够以一个for循环或while循环干净漂亮地完成下列动作（此种动作模式在面向对象程序方法中重要无比）：

```
CShape shapes[5];
... // 令 5 个 shapes 各为矩形、四方形、椭圆形、圆形、三角形
for (int i=0; i<5; i++)
{
    shapes[i].display;
}
```

5. Shape 只是一种抽象概念，世界上并没有「形状」这种东西！你可以在一个 C++程序中做以下动作，但是不符合生活法则：

```
CShape shape;    // 世界上没有「形状」这种东西，
shape.setcolor();// 所以这个动作就有点奇怪。
```

这同时也说出了第三点的另一个否定理由：按理你不能够把一个抽象的「形状」显示出来，不是吗?!

如果语法允许你产生一个不应该有的抽象对象，或如果语法不支持「把所有形状（不管什么形状）都 display 出来」的一般化动作，这就是个失败的语言。C++ 是成功的，自然有它的整治方式。记住，「面向对象」观念是描绘现实世界用的。所以，你可以以真实生活中的经验去思考程序设计的逻辑。

this 指针

刚刚我才说过，两个矩形对象rect1 和rect2 各有自己的m_color 成员变量，但rect1.setcolor 和rect2.setcolor却都通往唯一的 CRect::setcolor成 员 函 数 。

那么CRect::setcolor如何处理不同对象中的m_color？答案是：成员函数有一个隐藏参数，名为this指针。当你调用：

```
rect1.setcolor(2); // rect1 是 CRect 对象
rect2.setcolor(3); // rect2 是 CRect 对象
```

编译器实际上为你做出来的代码是：

```
CRect::setcolor(2, (CRect*)&rect1);
CRect::setcolor(3, (CRect*)&rect2);
```

不过，由于CRect本身并没有声明setcolor，它是从CShape继承来的，所以编译器实际上产生的代码是：

```
CShape::setcolor(2, (CRect*)&rect1);
CShape::setcolor(3, (CRect*)&rect2);
```

多出来的参数，就是所谓的 this 指针。至于类之中，成员函数的定义：

```
class CShape
{
...
public:
void setcolor(int color) { m_color = color; }
};
```

被编译器整治过后，其实是：

```
class CShape
{
...
public:
void setcolor(int color,(CShape*)this){ this->m_color = color; }
};
```

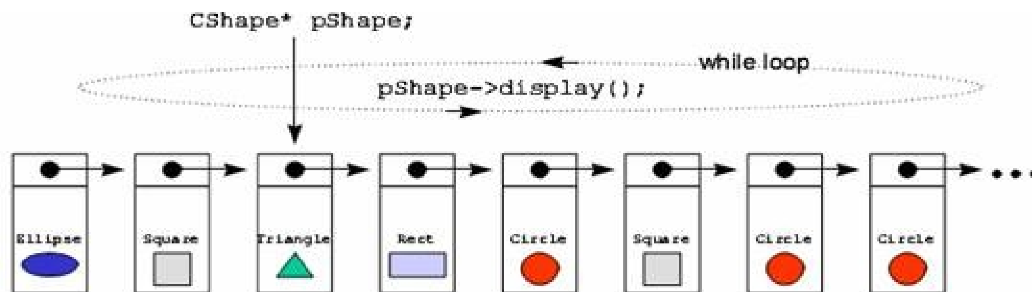
我们拨开了第一道疑云。

虚函数与多态(Polymorphism）

我曾经说过，前一个例子没有办法完成这样的动作：

```
CShape shapes[5];
... // 令 5 个 shapes 各为矩形、四方形、椭圆形、圆形、三角形
for (int i=0; i<5; i++)
{
shapes[i].display;
}
```

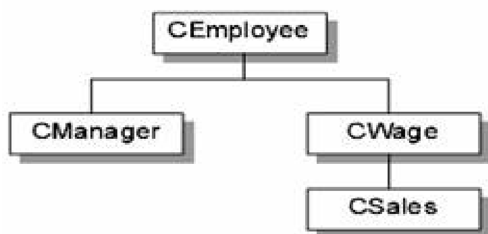
可是这种所谓对象操作的一般化动作在application framework中非常重要。作为framework设计者的我，总是希望能够准备一个display 函数，给我的使用者调用；不管他根据我的这一大堆形状类派生出其它什么奇形怪状的类，只要他想 display，像下面那么做就行。



为了支持这种能力，C++ 提供了所谓的虚函数（virtual function）。

虚拟 + 函数?! 听起来很恐怖的样子。如果你了解汽车的离合器踩下去代表汽车空档，空档表示失去引擎本身的牵制力，你就会了解「高速行驶间煞车绝不能踩离合器」的道理并矢志遵行。好，如果你真的了解为什么需要虚函数以及什么情况下需要它，你就能够掌握它的灵魂与内涵，真正了解它的设计原理，并且发现认为它非常人性。并且，真正知道怎么用它。

让我用另一个例子来展开我的说明。这个范例灵感得自Visual C++手册之一：Introdoction to C++。假设你的类种类如下：



程序代码实现如下：

```
#0001  #include <string.h>
#0003  //---------------------------------------------------------------
#0004  class CEmployee // 职员
#0005  {
#0006  private:
#0007      char m_name[30];
#0009  public:
#0010      CEmployee();
#0011      CEmployee(const char* nm) { strcpy(m_name, nm); }
#0012  };
#0013//---------------------------------------------------------------
#0014  class CWage : public CEmployee // 时薪职员是一种职员
#0015  {
#0016  private :
#0017      float m_wage;
#0018      float m_hours;
#0020  public :
#0021      CWage(const char* nm):CEmployee(nm){ m_wage = 250.0; m_hours =40.0; }
#0022      void setWage(float wg)  { m_wage = wg; }
#0023      void setHours(float hrs)  { m_hours = hrs; }
#0024      float computePay();
#0025  };
#0026//---------------------------------------------------------------
#0027  class CSales : public CWage // 销售员是一种时薪职员
#0028  {
#0029  private :
#0030      float m_comm;
#0031      float m_sale;
#0033  public :
#0034      CSales(const char* nm) : CWage(nm)  { m_comm = m_sale = 0.0; }
#0035      void setCommission(float comm)     {m_comm = comm; }
#0036      void setSales(float sale)          { m_sale = sale; }
#0037      float computePay();
#0038  };
#0039//---------------------------------------------------------------
#0040  class CManager : public CEmployee // 经理也是一种职员
#0041  {
#0042  private :
#0043      float m_salary;
#0044  public :
#0045      CManager(const char* nm):CEmployee(nm){ m_salary = 15000.0; }
#0046      void setSalary(float salary)          { m_salary = salary; }
#0047      float computePay();
#0048  };
#0049//---------------------------------------------------------------
#0050  void main()
#0051  {
#0052  CManager aManager("陈美静");
#0053  CSales   aSales("侯俊杰");
#0054  CWage    aWager("曾铭源");
#0055  }
#0056//---------------------------------------------------------------
#0057  //虽然各类的 computePay函数都没有定义，但因为程序也没有调用之，所以无妨。
```

如此一来，CWage继承了CEmployee所有的成员（包括数据与函数），CSales 又继承了CWage 所有的成员（包括数据与函数）。在意义上，相当于CSales拥有数据如下：

```
// private data of CEmployee
char m_name[30];
// private data of CWage
float m_wage;
float m_hours;
// private data of CSales
float m_comm;
float m_sale;
```

以及函数如下：

```
void setWage(float wg);
void setHours(float hrs);
void setCommission(float comm);
void setSale(float sales);
void computePay();
```

从Visual C++的除错器中，我们可以看到，上例的main执行之后，程序拥有三个对象，内容（我是指成员变量）分别为：



从薪水说起

虚函数的故事要从薪水的计算说起。根据不同职员的计薪方式，我设计 computePay 函数如下：

```
float CManager::computePay()
{
    return m_salary;  // 经理以「固定周薪」计薪。
}
float CWage::computePay()
{
    return (m_wage * m_hours); // 时薪职员以「钟点费 * 每周工时」计薪。
}
float CSales::computePay()
{   // 销售员以「钟点费 * 每周工时」再加上「佣金 * 销售额」计薪。
    return (m_wage * m_hours + m_comm * m_sale);  // 语法错误。
}
```

但是CSales对象不能够直接取用CWage的m_wage和m_hours，因为它们是 private成员变量。所以是不是应该改为这样：

```
float CSales::computePay()
{
    return computePay() + m_comm * m_sale;
}
```

这也不好，我们应该指明函数中所调用的 computePay 究归谁属——编译器没有厉害到能够自行判断而保证不出错。正确写法应该是：

```
float CSales::computePay()
{
    return CWage::computePay() + m_comm * m_sale;
}
```

这就合乎逻辑了：销售员是一般职员的一种，他的薪水应该是以时薪职员的计薪方式作为底薪，再加上额外的销售佣金。我们看看实际情况，如果有一个销售员：

```
CSales aSales("侯俊杰");
```

那么侯俊杰的底薪应该是:

```
aSales.CWage::computePay(); // 这是销售员的底薪。注意语法。
```

而侯俊杰的全薪应该是：

```
aSales.computePay(); // 这是销售员的全薪
```

结论是：要调用父类的函数，你必须使用scope resolution operator（::）明白指出。接下来我要触及对象类型的转换，这关系到指针的运用，更直接关系到为什么需要虚函数。了解它，对于application framework如MFC者的运用十分十分重要。

假设我们有两个对象：

```
CWage aWager;
CSales aSales("侯俊杰");
```

销售员是时薪职员之一，因此这样做是合理的：

```
aWager = aSales; // 合理，销售员必定是时薪职员。
```

这样就不合理：

```
aSales = aWager; // 错误，时薪职员未必是销售员。
```

如果你一定要转换，必须使用指针，并且明显地做类型转换（cast）动作：

```
CWage*  pWager;
CSales* pSales;
CSales  aSales("侯俊杰");
pWager = &aSales; // 把一个「基类指针」指向派生类之对象，合理且自然。
pSales = (CSales *)pWager; //强迫转型。语法上可以，但不符合现实生活。
```

真实世界中某些时候我们会以「一种动物」来总称猫啊、狗啊、兔子猴子等等。为了某种便利（这个便利稍后即可看到），我们也会想以「一个通用的指针」表示所有可能的职员类型。无论如何，销售员、时薪职员、经理，都是职员，所以下面动作合情合理：

```
CEmployee* pEmployee;
CWage aWager("曾铭源");
CSales aSales("侯俊杰");
CManager aManager("陈美静");
pEmpolyee = &aWager; // 合理，因为时薪职员必是职员
pEmpolyee = &aSales; // 合理，因为销售员必是职员
pEmpolyee = &aManager;  // 合理，因为经理必是职员
```

也就是说，你可以把一个「职员指针」指向任何一种职员。这带来的好处是程序设计的巨大弹性，譬如说你设计一个串列（linked list），各个元素都是职员（哪一种职员都可以），你的add函数可能因此希望有一个「职员指针」作为参数：

```
add(CEmployee* pEmp);  // pEmp 可以指向任何一种职员
```

晴天霹雳

我们渐渐接触问题的核心。上述 C++ 性质使真实生活经验的确在计算机语言中仿真了出来，但是万里无云的日子里却出现了一个晴天霹雳：如果你以一个「基类之指针」指向一个「派生类之对象」，那么经由此指针，你就只能够调用基类（而不是派生类）所定义的函数。因此：

```
CSales aSales("侯俊杰");
CSales* pSales;
CWage*  pWager;
pSales = &aSales;
pWager = &aSales; // 以「基类之指针」指向「派生类之对象」
pWager->setSales(800.0);//错误（编译器会检测出来），因为CWage并没有定义setSales 函数。
pSales->setSales(800.0);//正确，调用 CSales::setSales 函数。
```

虽然 pSales 和 pWager 指向同一个对象，但却因指针的原始类型而使两者之间有了差异。延续此例，我们看另一种情况：

```
pWager->computePay();  // 调用 CWage::computePay()
pSales->computePay();  // 调用 CSales::computePay()
```

虽然pSales和pWager实际上都指向CSales对象，但是两者调用的computePay 却不相同。到底调用到哪个函数，必须视指针的原始类型而定，与指针实际所指之对象无关。

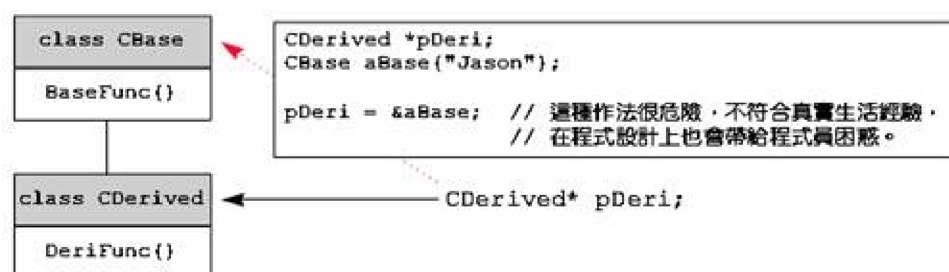
三个结论

我们得到了三个结论：

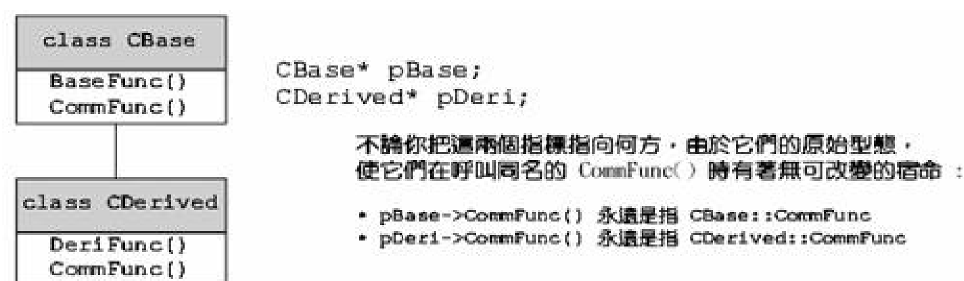
1. 如果你以一个「基类之指针」指向「派生类之对象」，那么经由该指针你只能够调用基类所定义的函数。



2. 如果你以一个「派生类之指针」指向一个「基类之对象」，你必须先做明显的转型动作（explicit cast）。这种作法很危险，不符合真实生活经验，在程序设计上也会带给程序员困惑。



3. 如果基类和派生类都定义了「相同名称之成员函数」，那么通过对象指针调用成员函数时，到底调用到哪一个函数，必须视该指针的原始类型而定，而不是视指针实际所指之对象的类型而定。这与第 1点其实意义相通。



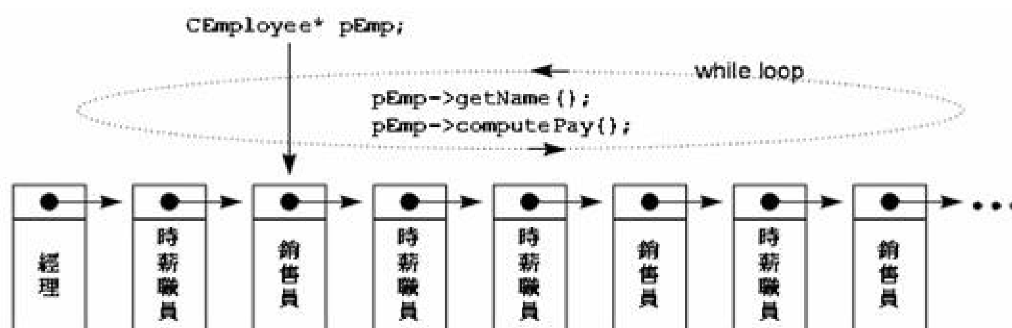
得到这些结论后，看看什么事情会困扰我们。前面我曾提到一个由职员组成的串列，如果我想写一个 printNames 函数走访串列中的每一个元素并印出职员的名字，我们可以在 CEmployee（最基类）中多加一个getName 函数，然后再设计一个while循环如下：

```
int count = 0;
CEmployee* pEmp;
...
while (pEmp = anIter.getNext())
{
    count++;
    cout << count << ' ' << pEmp->getName() << endl;
}
```

你可以把anIter.getNext想象是一个可以走访串列的函数，它传回 CEmPloyee*，也因此每一次获得的指针才可以调用定义于CEmployee中的 getName。

但是，由于函数的调用是依赖指针的原始类型而不管它实际上指向何方（何种对象），因此如果上述 while 循环中调用的是 pEmp->computePay，那么 while 循环所执行的将总是相同的运算，也就是 CEmployee::computePay，这就糟了（销售员领到经理的薪水还不糟吗）。更糟的是，我们根本没有定义 CEmployee::computePay，因为CEmployee只是个抽象概念（一个抽象类）。指针必须落实到实例类型上如CWage或CManager 或CSales，才有薪资计算公式。

计薪循环图



虚函数与一般化

我想你可以体会，上述的while 循环其实就是把动作「一般化」。「一般化」之所以重要，在于它可以把现在的、未来的情况统统纳入考虑。将来即使有另一种名曰「顾问」的职员，上述计薪循环应该仍然能够正常运作。当然啦，「顾问」的 computePay 必须设计好。

「一般化」是如此重要，解决上述问题因此也就迫切起来。我们需要的是什么呢？是能够「依旧以 CEmpolyee 指针代表每一种职员」，而又能够在「实际指向不同种类之职员」时，「调用到不同版本（不同类中）之 computePay」这种能力。

这种性质就是多态（polymorphism），靠虚函数来完成。

再次看看那张计薪循环图：

- 当pEmp指向经理，我希望pEmp->computePay是经理的薪水计算式，也就是 CManager::computePay。

- 当pEmp指向销售员，我希望pEmp->computePay是销售员的薪水计算式，也就是CSales::computePay。
- 当pEmp指向时薪职员，我希望pEmp->computePay是时薪职员的薪水计算式，

也就是CWage::computePay。

虚函数正是为了对「如果你以一个基类之指针指向一个派生类之对象，那么透过该指针你就只能够调用基类所定义之成员函数」这条规则反其道而行的设计。

不必设计复杂的串列函数如add或getNext才能验证这件事，我们看看下面这个简单例子。如果我把职员一例中所有四个类的computePay 函数前面都加上 virtual关键字，使它们成为虚函数，那么：

```
CEmployee*  pEmp;
CWage       aWager("曾铭源");
CSales      aSales("侯俊杰");
CManager    aManager("陈美静");
pEmp = &aWager;
cout << pEmp->computePay();  // 调用的是 CWage::computePay
pEmp = &aSales;
cout << pEmp->computePay();  // 调用的是 CSales::computePay
pEmp = &aManager;
cout << pEmp->computePay();  // 调用的是 CManager::computePay
```

现在重新回到Shape例子，我打算让display 成为虚函数：

```
#0001  #include <iostream.h>
#0002  class CShape
#0003  {
#0004   public:
#0005       virtual void display() { cout << "Shape \n"; }
#0006   };
#0007  //------------------------------------------------
#0008  class CEllipse : public CShape
#0009  {
#0010   public:
#0011       virtual void display() { cout << "Ellipse \n"; }
#0012   };
#0013  //------------------------------------------------
#0014  class CCircle : public CEllipse
#0015  {
#0016   public:
#0017       virtual void display() { cout << "Circle \n"; }
#0018  };
#0019  //------------------------------------------------
#0020  class CTriangle : public CShape
#0021  {
#0022   public:
#0023       virtual void display() { cout << "Triangle \n"; }
#0024  };
#0025  //------------------------------------------------
#0026  class CRect : public CShape
#0027  {
#0028   public:
#0029       virtual void display() { cout << "Rectangle \n"; }
#0030  };
#0031  //------------------------------------------------
#0032  class CSquare : public CRect
#0033  {
#0034   public:
#0035       virtual void display() { cout << "Square \n"; }
#0036  };
#0037  //------------------------------------------------
#0038  void main()
#0039  {
#0040   CShape       aShape;
#0041   CEllipse     aEllipse;
#0042   CCircle      aCircle;
#0043   CTriangle    aTriangle;
#0044   CRect        aRect;
#0045   CSquare      aSquare;
#0046   CShape* pShape[6] = { &aShape,&aEllipse,&aCircle,&aTriangle,&aRect, &aSquare };
#0053   for(int i=0; i< 6; i++)
#0054       pShape[i]->display();
#0055   }
#0056  //------------------------------------------------
```

得到的结果是：Shape\n Ellipse\n Circle \n Triangle\n Rectangle\n Square

如果把所有类中的virtual 保留字拿掉，执行结果变成：Shape Shape Shape Shape Shape Shape

综合Employee 和Shape两例，第一个例子是：

```
pEmp = &aWager;
cout << pEmp->computePay();
pEmp = &aSales;
cout << pEmp->computePay();
pEmp = &aBoss;                  这三进程序代码完全相同
cout << pEmp->computePay();
```

第二个例子是：

```
CShape* pShape[6];
for (int i=0; i< 6; i++)
pShape[i]->display();  // 此进程序代码执行了 6 次。
```

我们看到了一种奇特现象：程序代码完全一样（因为一般化了），执行结果却不相同。这就是虚函数的妙用。

如果没有虚函数这种东西，你还是可以使用 scope resolution operator（::）明白指出调用哪一个函数，但程序就不再那么优雅与弹性了。

从操作型定义来看，什么是虚函数呢？如果你预期派生类有可能重新定义某一个成员函数，那么你就在基类中把此函数设为 virtual。MFC 有两个十分十分重要的虚函数：与 document 有关的 Serialize 函数和与 view 有关的 OnDraw 函数。你应该在自己的 CMyDoc 和 CMyView 中改写这两个虚函数。

多态（Polymorphism）

你看，我们以相同的指令却唤起了不同的函数，这种性质称为Polymorphism，意思是 "the ability to assume many forms"（多态）。编译器无法在编译时期判断 pEmp->computePay到底是调用哪一个函数，必须在运行时才能评估之，这称为后期联编 late binding 或动态联编 dynamic binding。至于 C 函数或 C++ 的 non-virtual 函数，在编译时期就转换为一个固定地址的调用了，这称为前期联编 early binding 或静态联编 static binding。

Polymorphism 的目的，就是要让处理「基类之对象」的程序代码，能够完全透通地继续适当处理「派生类之对象」。

可以说，虚函数是了解多态（Polymorphism）以及动态联编的关键。同时，它也是了解如何使用 MFC 的关键。

让我再次提示你，当你设计一套类，你并不知道使用者会派生什么新的子类出来。如果动物世界中出现了新品种名曰雅虎，类使用者势必在 CAnimal 之下派生一个CYahoo。饶是如此，身为基类设计者的你，可以利用虚函数的特性，将所有动物必定会有的行为（例如哮叫 roar），规划为虚函数，并且规划一些一般化动作（例如「让每一种动物发出一声哮叫」）。那么，虽然，你在设计基类以及这个一般化动作时，无法掌握使用者自行派生的子类，但只要他改写了 roar 这个虚函数，你的一般化对象操作动作自然就可以调用到该函数。

再次回到前述的Shape例子。我们说CShape是抽象的，所以它根本不该有 display这个动作。但为了在各实例派生类中绘图，我们又不得不在基类 CShape 加上display 虚函数。你可以定义它什么也不做（空函数）：

```
class CShape
{
public:
    virtual void display() {  }
};
或只是给个消息：
class CShape
{
public:
    virtual void display() { cout << "Shape \n"; }
};
```

这两种作法都不高明，因为这个函数根本就不应该被调用（CShape 是抽象的），我们根本就不应该定义它。不定义但又必须保留一块空间（spaceholder）给它，于是 C++ 提供了所谓的纯虚函数：

```
class CShape
{
public:
    virtual void display() = 0;  // 注意 "= 0"
};
```

纯虚函数不需定义其实际动作，它的存在只是为了在派生类中被重新定义，只是为了提供一个多态接口。只要是拥有纯虚函数的类，就是一种抽象类，它是不能够被实例化（instantiate）的，也就是说，你不能根据它产生一个对象（你怎能说一种形状为 'Shape' 的物体呢）。如果硬要强渡关山，会换来这样的编译消息：

```
error : illegal attempt to instantiate abstract class.
```

关于抽象类，我还有一点补充。CCircle 继承了CShape 之后，如果没有改写CShape中的纯虚函数，那么CCircle 本身也就成为一个拥有纯虚函数的类，于是它也是一个抽象类。

是对虚函数做结论的时候了：

- 如果你期望派生类重新定义一个成员函数，那么你应该在基类中把此函数设为virtual。
- 以单一指令唤起不同函数，这种性质称为Polymorphism，意思是 "the ability to assume many forms"，也就是多态。
- 虚函数是C++语言的Polymorphism 性质以及动态联编的关键。
- 既然抽象类中的虚函数不打算被调用，我们就不应该定义它，应该把它设为纯虚函数（在函数声明之后加上"=0" 即可）。
- 我们可以说，拥有纯虚函数者为抽象类（abstract Class），以别于所谓的实例类（concrete class)。

抽象类不能产生出对象实体，但是我们可以拥有指向抽象类之指针，以便于操作抽象类的各个派生类。

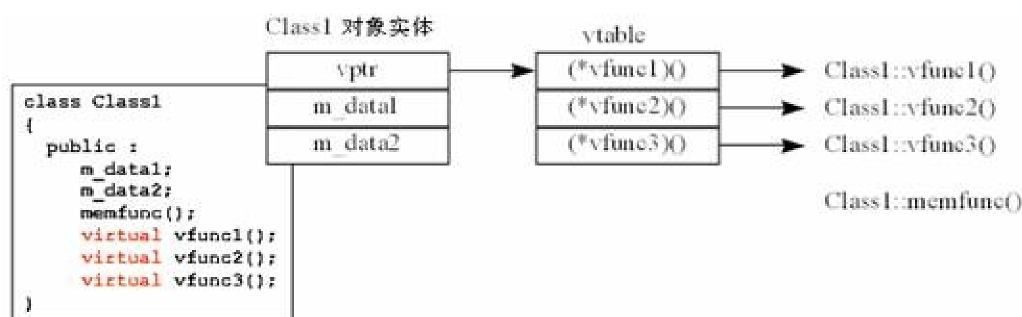
虚函数派生下去仍为虚函数，而且可以省略virtual关键词。

类与对象大解剖

为了达到动态联编（后期联编）的目的，C++ 编译器通过某个表格，在运行时「间接」调用实际上欲联编的函数（注意「间接」这个字眼）。这样的表格称为虚函数表（常被称为 vtable）。每一个「内含虚函数的类」，编译器都会为它做出一个虚函数表，表中的每一个元素都指向一个虚函数的地址。此外，编译器当然也会为类加上一项成员变量，是一个指向该虚函数表的指针（常被称为 vptr）。举个例：

```
class Class1  {
public :
    data1;
    data2;
    memfunc();
    virtual vfunc1();
    virtual vfunc2();
    virtual vfunc3();
};
```

Class1 对象实体在内存中占据这样的空间：



C++ 类的成员函数，你可以想象就是C语言中的函数。它只是被编译器改过名称，并增加一个参数（this 指针），因而可以处理调用者（C++ 对象）中的成员变量。所以，你并没有在 Class1 对象的内存区块中看到任何与成员函数有关的任何东西。

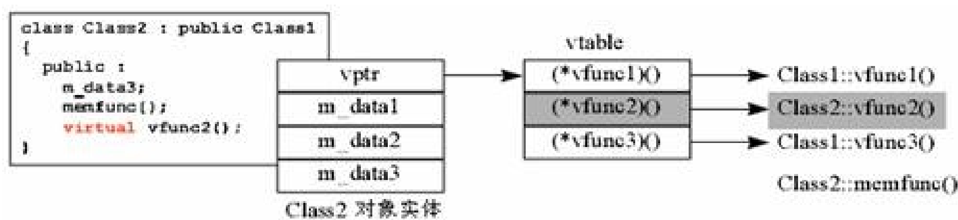
每一个由此类派生出来的对象，都有这么一个 vptr。当我们通过这个对象调用虚函数，事实上是通过 vptr 找到虚函数表，再找出虚函数的真正地址。

奥妙在于这个虚函数表以及这种间接调用方式。虚函数表的内容是依据类中的虚函数声明次序，一一填入函数指针。派生类会继承基类的虚函数表（以及所有其它可以继承的成员），当我们在派生类中改写虚函数时，虚函数表就受了影响：表中元素所指的函数地址将不再是基类的函数地址，而是派生类的函数地址。

看看这个例子：

```
class Class2 : public Class1 {
public :
    data3;
    memfunc();
    virtual vfunc2();
};
```

于是，一个「指向Class1所生对象」的指针，所调用的 vfunc2 就是 Class1::vfunc2，而一个「指向Class2 所生对象」的指针，所调用的 vfunc2 就是 Class2::vfunc2。动态联编机制，在运行时，根据虚函数表，做出了正确的选择。我们解开了第二道神秘。口说无凭，何不看点实际。观其地址，物焉廋哉，下面是一个测试程序：



```
#0001  #include <iostream.h>
#0002  #include <stdio.h>
#0004  class ClassA
#0005  {
#0006  public:
#0007  int m_data1;
#0008  int m_data2;
#0009  void func1() { }
#0010  void func2() { }
#0011  virtual void vfunc1() { }
#0012  virtual void vfunc2() { }
#0013  };
#0015  class ClassB : public ClassA
#0016  {
#0017  public:
#0018  int m_data3;
#0019  void func2() {  }
#0020  virtual void vfunc1() { }
#0021  };
#0023  class ClassC : public ClassB
#0024  {
#0025  public:
#0026  int m_data1;
#0027  int m_data4;
#0028  void func2() {  }
#0029  virtual void vfunc1() { }
#0030  };
#0032  void main()
#0033  {
#0034     cout << sizeof(ClassA) << endl;
#0035     cout << sizeof(ClassB) << endl;
#0036     cout << sizeof(ClassC) << endl;
#0038     ClassA a;
#0039     ClassB b;
#0040     ClassC c;
#0041
#0042     b.m_data1 = 1;
#0043     b.m_data2 = 2;
#0044     b.m_data3 = 3;
#0045     c.m_data1 = 11;
#0046     c.m_data2 = 22;
#0047     c.m_data3 = 33;
#0048     c.m_data4 = 44;
#0049     c.ClassA::m_data1 = 111;
#0050
#0051     cout << b.m_data1 << endl;
#0052     cout << b.m_data2 << endl;
#0053     cout << b.m_data3 << endl;
#0054     cout << c.m_data1 << endl;
#0055     cout << c.m_data2 << endl;
#0056     cout << c.m_data3 << endl;
#0057     cout << c.m_data4 << endl;
#0058     cout << c.ClassA::m_data1 << endl;
#0059
#0060     cout << &b << endl;
#0061     cout << &(b.m_data1) << endl;
#0062     cout << &(b.m_data2) << endl;
#0063     cout << &(b.m_data3) << endl;
#0064     cout << &c << endl;
#0065     cout << &(c.m_data1) << endl;
#0066     cout << &(c.m_data2) << endl;
#0067     cout << &(c.m_data3) << endl;
#0068     cout << &(c.m_data4) << endl;
#0069     cout << &(c.ClassA::m_data1) << endl;
#0070  }
```

执行结果与分析如下：

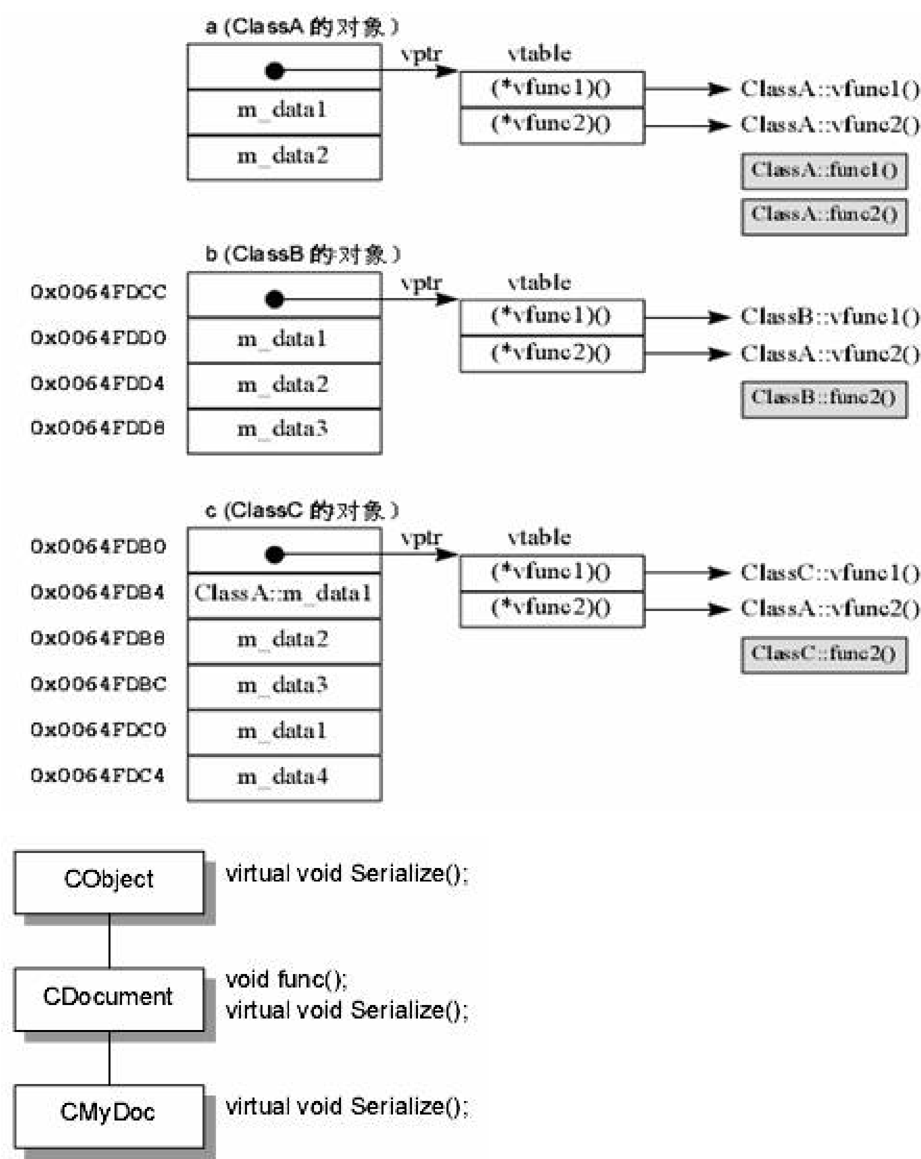
执行结果	意义	说明
12	Sizeof (ClassA)	2 个int 加上一个vptr
16	Sizeof (ClassB)	继承自ClassA，再加上1 个int
24	Sizeof (ClassC)	继承自ClassB，再加上2 个int
1	b.m_data1 的内容	
2	b.m_data2 的内容	
3	b.m_data3 的内容	
11	c.m_data1 的内容	
22	c.m_data2 的内容	
33	c.m_data3 的内容	
44	c.m_data4 的内容	
111	c.ClassA::m_data1的内容	
0x0064FDCC	b 对象的起始地址	这个地址中的内容就是vptr
0x0064FDD0	b.m_data1 的地址	
0x0064FDD4	b.m_data2 的地址	
0x0064FDD8	b.m_data3 的地址	
0x0064FDB0	c 对象的起始地址	这个地址中的内容就是vptr
0x0064FDC0	c.m_data1 的地址	
0x0064FDB8	c.m_data2 的地址	
0x0064FDBC	c.m_data3 的地址	
0x0064FDC4	c.m_data4 的地址	
0x0064FDB4	c.ClassA::m_data1 的地址	

a、b、c 对象的内容图标如下：

Object slicing与虚函数

我要在这里说明虚函数另一个极重要的行为模式。假设有三个类，阶层关系如下：

以程序表现如下：


a (ClassA 的对象)
vptr
vtable
m_data1
m_data2
(*vfunc1)()
(*vfunc2)()
ClassA::vfunc1()
ClassA::vfunc2()
ClassA::func1()
ClassA::func2()
b (ClassB 的对象)
0x0064FDCC
0x0064FDD0
0x0064FDD4
0x0064FDD8
m_data1
m_data2
m_data3
ClassB::vfunc1()
ClassA::vfunc2()
ClassB::func2()
c (ClassC 的对象)
0x0064FDB0
0x0064FDB4
0x0064FDB8
0x0064FDBC
0x0064FDC0
0x0064FDC4
ClassA::m_data1
m_data2
m_data3
m_data1
m_data4
ClassC::vfunc1()
ClassA::vfunc2()
ClassC::func2()
CObject
virtual void Serialize();
CDocument
void func();
virtual void Serialize();
CMyDoc
virtual void Serialize();

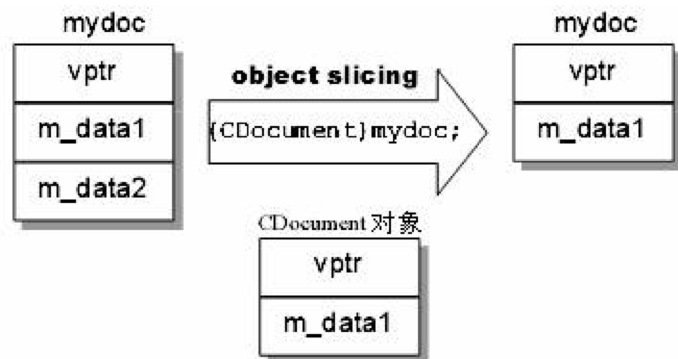
```
#0001  #include <iostream.h>
#0003  class CObject
#0004  {
#0005   public:
#0006       virtual void Serialize() { cout << "CObject::Serialize() \n\n"; }
#0007  };
#0009  class CDocument : public CObject
#0010  {
#0011  public:
#0012      int m_data1;
#0013      void func() { cout << "CDocument::func()" << endl;
#0014                    Serialize();
#0015                  }
#0017  virtual void Serialize(){cout << "CDocument::Serialize() \n\n"; }
#0018  };
#0020  class CMyDoc : public CDocument
#0021  {
#0022  public:
#0023      int m_data2;
#0024      virtual void Serialize() { cout << "CMyDoc::Serialize() \n\n"; }
#0025  };
#0026 //---------------------------------------------------------------
#0027  void main()
#0028  {
#0029  CMyDoc  mydoc;
#0030  CMyDoc* pmydoc = new CMyDoc;
#0032  cout << "#1 testing" << endl;
#0033  mydoc.func();
#0035  cout << "#2 testing" << endl;
#0036  ((CDocument*)(&mydoc))->func();
#0038  cout << "#3 testing" << endl;
#0039  pmydoc->func();
#0041  cout << "#4 testing" << endl;
#0042  ((CDocument)mydoc).func();
#0043  }
```

由于CMyDoc自己没有func 函数，而它继承了CDocument 的所有成员，所以 main 之中的四个调用动作毫无问题都是调用CDocument::func。但，CDocument::func 中所调用Serialize 是哪一个类的成员函数呢？如果它是一般（non-virtual）函数，毫无问题应该是CDocument::Serialize。但因为这是个虚函数，情况便有不同。以下是执行结果：

```
#1 testing
CDocument::func()
CMyDoc::Serialize()
#2 testing
CDocument::func()
CMyDoc::Serialize()
#3 testing
CDocument::func()
CMyDoc::Serialize()
#4 testing
CDocument::func()
CDocument::Serialize()  <-- 注意
```

前三个测试都符合我们对虚函数的期望：既然派生类已经改写了虚函数 Serialize，那么理当调用派生类之Serialize 函数。这种行为模式非常频繁地出现在applicationframework 身上。后续当我追踪 MFC 原始代码时，遇此情况会再次提醒你。

第四项测试结果则有点出乎意料之外。你知道，派生对象通常都比基础对象大（我是指内存空间），因为派生对象不但继承其基类的成员，又有自己的成员。那么所谓的upcasting（向上强制转型）：(CDocument)mydoc，将会造成对象的内容被切割（object slicing）：



当我们调用：((CDocument)mydoc).func(); mydoc已经是一个被切割得剩下半条命的对象，而 func 内部调用虚函数 Serialize；后者将使用的「mydoc 的虚函数指针」虽然存在，它的值是什么呢？你是不是隐隐觉得有什么大灾难要发生？

幸运的是，由于 ((CDocument)mydoc).func() 是个传值而非传址动作，编译器以所谓的复制构造函数（copy constructor）把CDocument 对象内容复制了一份，使得mydoc的vtable 内容与CDocument对象的vtable 相同。本例虽没有明显做出一个复制构造函数，编译器会自动为你合成一个。

说这么多，总结就是，经过所谓的data slicing，本例的mydoc 真正变成了一个完完全全的CDocument 对象。所以，本例的第四项测试结果也就水落石出了。注意，"upcasting" 并不是惯用的动作，应该小心，甚至避免。

静态成员（变量与函数）

我想你已经很清楚了，如果你依据一个类产生出三个对象，每一个对象将各有一份成员变量。有时候这并不是你要的。假设你有一个类，专门用来处理存款账户，它至少应该要有存户的姓名、地址、存款额、利率等成员变量：

```
class SavingAccount
{
private:
    char m_name[40];  // 存户姓名
    char m_addr[60];  // 存户地址
    double m_total; // 存款额
    double m_rate; // 利率
    ...
};
```

这家行库采用浮动利率，每个账户的利息都是根据当天的挂牌利率来计算。这时候m_rate 就不适合成为每个账户对象中的一笔数据，否则每天一开市，光把所有账户内容叫出来，修改 m_rate 的值，就花掉不少时间。m_rate 应该独立在各对象之外，成为类独一无二的数据。怎么做？在 m_rate 前面加上 static 修饰词即可：

```
class SavingAccount
{
private:
    char m_name[40]; // 存户姓名
    char m_addr[60]; // 存户地址
    double m_total; // 存款额
    static double m_rate; // 利率
    ...
};
```

static 成员变量不属于对象的一部分，而是类的一部分，所以程序可以在还没有诞生任何对象的时候就处理此种成员变量。但首先你必须初始化它。

不要把 static 成员变量的初始化动作安排在类的构造函数中，因为构造函数可能一再被调用，而变量的初值却只应该设定一次。也不要把初始化动作安排在头文件中，因为它可能会被含入许多地方，因此也就可能被执行许多次。你应该在实现文件中且类以外的任何位置设定其初值。例如在 main 之中，或全局函数中，或任何函数之外：

```
double SavingAccount::m_rate = 0.0075;  // 设立 static 成员变量的初值
void main() { ... }
```

这么做可曾考虑到 m_rate 是个 private 数据？没关系，设定 static 成员变量初值时，不受任何存取权限的束缚。请注意，static 成员变量的类型也出现在初值设定句中，因为这是一个初值设定动作，不是一个数量指定（assignment）动作。事实上，static 成员变量是在这时候（而不是在类声明中）才定义出来的。如果你没有做这个初始化动作，会产生链接错误：

```
error LNK2001: unresolved external symbol "private: static double
SavingAccount::m_rate"(?m_rate@SavingAccount@@2HA)
```

下面是存取static成员变量的一种方式，注意，此刻还没有诞生任何对象实体：

```
// 第一种存取方式
void main()
{
    SavingAccount::m_rate = 0.0075; // 欲此行成立，须把 m_rate 改为 public
}
下面这种情况则是产生一个对象后，通过对象来处理static成员变量：
// 第二种存取方式
void main()
{
    SavingAccount myAccount;
    myAccount.m_rate = 0.0075;  // 欲此行成立，须把 m_rate 改为 public
}
```

你得搞清楚一个观念，static 成员变量并不是因为对象的实现而才得以实现，它本来就存在，你可以想象它是一个全局变量。因此，第一种处理方式在意义上比较不会给人错误的印象。

只要access level允许，任何函数（包括全局函数或成员函数，static 或 non-static）都可以存取static成员变量。但如果你希望在产生任何object之前就存取其class的private static 成员变量，则必须设计一个static成员函数（例如以下的 setRate）：

```
class SavingAccount
{
private:
    char m_name[40];  // 存户姓名
    char m_addr[60];  // 存户地址
    double m_total;   // 存款额
    static double m_rate; // 利率
    ...
public:
    static void setRate(double newRate) { m_rate = newRate; }
    ...
};
double SavingAccount::m_rate = 0.0075;  // 设立 static 成员变量的初值
void main()
{
    SavingAccount::setRate(0.0074);  // 直接调用类的 static 成员函数
    SavingAccount myAccount;
    myAccount.setRate(0.0074);        // 通过对象调用 static 成员函数
}
```

由于static成员函数不需要借助任何对象，就可以被调用执行，所以编译器不会为它暗加一个this指针。也因为如此，static成员函数无法处理类之中的 non-static 成员变量。还记得吗，我在前面说过，成员函数之所以能够以单一一份函数代码处理各个对象的数据而不紊乱，完全靠的是this指针的指示。

static 成员函数「没有this 参数」的这种性质，正是我们的MFC应用程序在准备callback函数时所需要的。第 6 章的 Hello World 例中我就会举这样一个实例。

C++ 程序的生与死：兼谈构造函数与析构函数

C++ 的new运算符和C的malloc函数都是为了配置内存，但前者比之后者的优点是，new不但配置对象所需的内存空间时，同时会引发构造函数的执行。

所谓构造函数（constructor），就是对象诞生后第一个执行（并且是自动执行）的函数，它的函数名称必定要与类名称相同。

相对于构造函数，自然就有个析构函数（destructor），也就是在对象行将毁灭但未毁灭之前一刻，最后执行（并且是自动执行）的函数，它的函数名称必定要与类名称相同，再在最前面加一个~符号。

一个有着阶层架构的类群组，当派生类的对象诞生之时，构造函数的执行是由最基类（most based）至最尾端派生类（most derived）；当对象要毁灭之前，析构函数的执行则是反其道而行。第 3 章的 frame1 程序对此有所示范。

我以实例展示不同种类之对象的构造函数执行时机。程序代码中的编号请对照执行结果。

```
#0001  #include <iostream.h>
#0002  #include <string.h>
#0004  class CDemo
#0005  {
#0006  public:
#0007      CDemo(const char* str);
#0008      ~CDemo();
#0009  private:
#0010      char name[20];
#0011  };
#0013  CDemo::CDemo(const char* str) // 构造函数
#0014  {
#0015      strncpy(name, str, 20);
#0016      cout << "Constructor called for " << name << '\n';
#0017  }
#0019  CDemo::~CDemo()// 析构函数
#0020  {
#0021      cout << "Destructor called for " << name << '\n';
#0022  }
#0024  void func()
#0025  {
#0026 CDemo LocalObjectInFunc("LocalObjectInFunc");  // in stack  ⑤
#0027 static CDemo StaticObject("StaticObject");     // local static ⑥
#0028 CDemo* pHeapObjectInFunc=new CDemo("HeapObjectInFunc");//in heap  ⑦
#0030 cout << "Inside func" << endl;  ⑧
#0032  }  ⑨
#0034  CDemo GlobalObject("GlobalObject");  // global static ①
#0036  void main()
#0037  {
#0038 CDemo LocalObjectInMain("LocalObjectInMain"); // in stack ②
#0039 CDemo* pHeapObjectInMain=new CDemo("HeapObjectInMain");//in heap ③
#0041 cout << "In main, before calling func\n";④
#0042 func();
#0043 cout << "In main, after calling func\n";  ❿
#0045  }  ①  ②  ③
```

以下是执行结果：

```
① Constructor called for GlobalObject
② Constructor called for LocalObjectInMain
③ Constructor called for HeapObjectInMain
④ In main, before calling func
⑤ Constructor called for LocalObjectInFunc
⑥ Constructor called for StaticObject
⑦ Constructor called for HeapObjectInFunc
⑧ Inside func
⑨ Destructor called for LocalObjectInFunc
❿ In main, after calling func
① Destructor called for LocalObjectInMain
② Destructor called for StaticObject
③ Destructor called for GlobalObject
```

我的结论是：

对于全局对象（如本例之GlobalObject），程序一开始，其构造函数就先被执行（比程序进入点更早）；程序即将结束前其析构函数被执行。MFC 程序就有这样一个全局对象，通常以 application object 称呼之，你将在第 6 章看到它。

对于局部对象，当对象诞生时，其构造函数被执行；当程序流程将离开该对象的存活范围（以至于对象将毁灭），其析构函数被执行。

对于静态（static）对象，当对象诞生时其构造函数被执行；当程序将结束时（此对象因而将遭致毁灭）其析构函数才被执行，但比全局对象的析构函数早一步执行。

对于以new方式产生出来的局部对象，当对象诞生时其构造函数被执行。析构式则在对象被delete 时执行（上例程序未示范）。

四种不同的对象生存方式（in stack、in heap、global、local static）

既然谈到了static对象，就让我把所有可能的对象生存方式及其构造函数调用时机做个整理。所有作法你都已经在前一节的小程序中看过。

在C++ 中，有四种方法可以产生一个对象。第一种方法是在堆栈（stack）之中产生它：

```
void MyFunc()
{
    CFoo foo;  // 在堆栈（stack）中产生 foo 对象
    ...
}
```

第二种方法是在堆积（heap）之中产生它：

```
void MyFunc()
{
    ...
    CFoo* pFoo = new CFoo(); // 在堆积（heap）中产生对象
}
```

第三种方法是产生一个全局对象（同时也必然是个静态对象）：

```
CFoo foo; // 在任何函数范围之外做此动作
```

第四种方法是产生一个局部静态对象：

```
void MyFunc()
{
    static CFoo foo;  // 在函数范围（scope）之内的一个静态对象
    ...
}
```

不论任何一种作法，C++ 都会产生一个针对CFoo 构造函数的调用动作。前两种情况，C++ 在配置内存 -- 来自堆栈（stack）或堆积（heap）-- 之后立刻产生一个隐藏的（你的原始代码中看不出来的）构造函数调用。第三种情况，由于对象实现于任何「函数活动范围（function scope）」之外，显然没有地方来安置这样一个构造函数调用动作。

是的，第三种情况（静态全局对象）的构造函数调用动作必须靠startup代码帮忙。startup代码是什么？是更早于程序进入点（main 或 WinMain）执行起来的代码，由 C++ 编译器提供，被链接到你的程序中。startup 代码可能做些像函数库初始化、进程信息设立、I/O stream 产生等等动作，以及对 static 对象的初始化动作（也就是调用其构造函数）。

当编译器编译你的程序，发现一个静态对象，它会把这个对象加到一个串列之中。更精确地说则是，编译器不只是加上此静态对象，它还加上一个指针，指向对象之构造函数及其参数（如果有的话）。把控制权交给程序进入点（main 或 WinMain）之前，startup代码会快速在该串列上移动，调用所有登记有案的构造函数并使用登记有案的参数，于是就初始化了你的静态对象。

第四种情况（局部静态对象）相当类似C语言中的静态局部变量，只会有一个实体（instance）产生，而且在固定的内存上（既不是stack 也不是heap）。它的构造函数在控制权第一次移转到其声明处（也就是在MyFunc第一次被调用）时被调用。

所谓 "Unwinding"

C++ 对象依其生存空间，适当地依照一定的顺序被析构（destructed）。但是如果发生异常情况（exception），而程序设计了异常情况处理程序（exception handling），控制权就会截弯取直地「直接跳」到你所设定的处理例程去，这时候堆栈中的 C++ 对象有没有机会被析构？这得视编译器而定。如果编译器有支持 unwinding 功能，就会在一个异常情况发生时，将堆栈中的所有对象都析构掉。

关于异常情况（exception）及异常处理（exception handling），稍后有一节讨论之。

运行时类型信息（RTTI）

我们有可能在程序执行过程中知道某个对象是属于哪一种类吗？这种在C++ 中称为运行时类型信息（Runtime Type Information，RTTI）的能力，晚近较先进的编译器如Visual C++ 4.0和Borland C++ 5.0 才开始广泛支持。以下是一个实例：

```
#0001  // RTTI.CPP - built by   C:\> cl.exe -GR rtti.cpp <ENTER>
#0002  #include <typeinfo.h>
#0003  #include <iostream.h>
#0004  #include <string.h>
#0006  class graphicImage
#0007  {
#0008   protected:
#0009   char name[80];
#0011   public:
#0012       graphicImage()
#0013       {
#0014           strcpy(name,"graphicImage");
#0015       }
#0017       virtual void display()
#0018       {
#0019           cout << "Display a generic image." << endl;
#0020       }
#0022       char* getName()
#0023       {
#0024           return name;
#0025       }
#0026  };
#0027 //--------------------------------------------------------------
#0028  class GIFimage : public graphicImage
#0029  {
#0030   public:
#0031       GIFimage()
#0032      {
#0033           strcpy(name,"GIFimage");
#0034      }
#0036       void display()
#0037       {
#0038           cout << "Display a GIF file." << endl;
#0039       }
#0040  };
#0042  class PICTimage : public graphicImage
#0043  {
#0044   public:
#0045       PICTimage()
#0046       {
#0047           strcpy(name,"PICTimage");
#0048       }
#0050       void display()
#0051       {
#0052           cout << "Display a PICT file." << endl;
#0053       }
#0054  };
#0055 //-------------------------------------------------------------
#0056  void processFile(graphicImage *type)
#0057  {
#0058      if (typeid(GIFimage) == typeid(*type))
#0059      {
#0060          ((GIFimage *)type)->display();
#0061      }
#0062      else if (typeid(PICTimage) == typeid(*type))
#0063      {
#0064          ((PICTimage *)type)->display();
#0065      }
#0066      else
#0067         cout << "Unknown type! " << (typeid(*type)).name() << endl;
#0068  }
#0070  void main()
#0071  {
#0072      graphicImage *gImage = new GIFimage();
#0073      graphicImage *pImage = new PICTimage();
#0075      processFile(gImage);
#0076      processFile(pImage);
#0077  }
```

执行结果如下：

```
Display a GIF file.
Display a PICT file.
```

这个程序与RTTI相关的地方有三个：

1. 编译时需选用/GR 选项（/GR 的意思是enable C++ RTTI）

2. 含入typeinfo.h

3. 新的typeid 运算符。这是一个重载（overloading）运算符，重载的意思就是拥有一个以上的型式，你可以想象那是一种静态的多态（Polymorphism）。typeid的参数可以是类名称（如本例#58 左），也可以是对象指针（如本例#58 右）。它传回一个type_info&。type_info 是一个类，定义于typeinfo.h 中：

```
class type_info {
public:
    virtual ~type_info();
    int operator==(const type_info& rhs) const;
    int operator!=(const type_info& rhs) const;
    int before(const type_info& rhs) const;
    const char* name() const;
    const char* raw_name() const;
private:
    ...
};
```

虽然Visual C++编译器自从4.0 版已经支持RTTI，但MFC 4.x并未使用编译器的能力完成其对RTTI 的支持。MFC有自己一套沿用已久的办法（从1.0 版就开始了）。喔，不要因为 MFC 的作法特殊而非难它，想想看它的悠久历史。

MFC的RTTI能力牵扯到一组非常神秘的宏（DECLARE_DYNAMIC、IMPLEMENT_DYNAMIC）和一个非常神秘的类（CRuntimeClass）。MFC 程序员都知道怎么用它，却没几个人懂得其运作原理。大道不过三两行，说穿不值一文钱，下一章我就模拟出一个 RTTI 的DOS 版本给你看。

动态生成（Dynamic Creation）

面向对象术语中有一个名为persistence，意思是永续存留。放在RAM中的东西，生命受到电力的左右，不可能永续存留；唯一的办法是把它写到文件去。MFC 的一个术语Serialize，就是做有关文件读写的永续存留动作，并且实做作出一个虚函数，就叫作Serialize。

看起来永续存留与本节的主题「动态生成」似乎没有什么干连。有！你把你的数据储存到文件，这些数据很可能（通常是）对象中的成员变量；我把它读出来后，势必要依据文件上的记载，重新new 出那些个对象来。问题在于，即使我的程序有那些类定义（就算我的程序和你的程序有一样的内容好了），我能够这么做吗：

```
char className[30] = getClassName();  // 从文件（或使用者输入）获得一个类名称
CObject* obj = new classname;         // 这一行行不通
```

首先，new classname 这个动作就过不了关。其次，就算过得了关，new 出来的对象究竟该是什么类类型？虽然以一个指向MFC 类老祖宗（CObject）的对象指针来容纳它绝对没有问题，但终不好总是如此吧！不见得这样子就能够满足你的程序需求啊。

显然，你能够以Serialize函数写文件，我能够以Serialize函数读文件，但我就是没办法恢复你原来的状态—— 除非我的程序能够「动态生成」。

MFC支持动态生成，靠的是一组非常神秘的宏（ DECLARE_DYNCREATE 、 IMPLEMENT_DYNCREATE）和一个非常神秘的类（CRuntimeClass）。第 3 章中我将把它抽丝剥茧，以一个 DOS 程序仿真出来。

异常处理（Exception Handling）

Exception（异常情况）是一个颇为新鲜的 C++ 语言特征，可以帮助你管理运行时的错误，特别是那些发生在深度巢状（nested）函数调用之中的错误。Watcom C++ 是最早支持ANSI C++异常情况的编译器，Borland C++ 4.0随后跟进，然后是 Microsoft Visual C++ 和 Symantec C++。现在，这已成为 C++ 编译器必需支持的项目。

C++ 的exception 基本上是与C的setjmp和longjmp函数对等的东西，但它增加了一些功能，以处理 C++ 程序的特别需求。从深度巢状的例程调用中直接以一条快捷方式撤回到异常情况处理例程（exception handler），这种「错误管理方式」远比结构化程序中经过层层的例程传回一系列的错误状态来的好。事实上exception handling是MFC和OWL两个application frameworks 的防弹中心。

C++ 导入了三个新的 exception 保留字：

1.try。之后跟随一段以{ }圈出来的程序代码，exception可能在其中发生。

2.catch。之后跟随一段以{ } 圈出来的程序代码，那是exception 处理例程之所在。catch 应该紧跟在 try 之后。

3.throw。这是一个指令，用来产生（丢出）一个exception。

下面是个实例：

```
try {
    // try block.
}
catch (char *p) {
    printf("Caught a char* exception, value %s\n",p);
}
catch (double d) {
    printf("Caught a numeric exception, value %g\n",d);
}
catch (...) {  // catch anything
    printf("Caught an unknown exception\n");
}
```

MFC早就支持exception，不过早期它用的是非标准语法。Visual C++ 4.0 编译器本身支持完整的C++ exceptions，MFC也因此有了两个exception 版本：你可以使用语言本身提供的性能，也可以沿用MFC古老的方法（以宏形式出现）。人们曾经因为MFC 的方案不同于ANSI标准而非难它，但是不要忘记它已经运作了多少年。

MFC的exceptions机制是以宏和exception types为基础。这些宏类似C++ 的exception 保留字，动作也满像。MFC以下列宏仿真C++ exception handling：

```
TRY
CATCH(type,object)
AND_CATCH(type,object)
END_CATCH
CATCH_ALL(object)
AND_CATCH_ALL(object)
END_CATCH_ALL
END_TRY
THROW()
THROW_LAST()
```

MFC所使用的语法与日渐浮现的标准稍微不同，不过其间差异微不足道。为了以MFC捕捉exceptions，你应该建立一个TRY 区块，下面接着CATCH 区块：

```
TRY {
    // try block.
}
CATCH (CMemoryException, e) {
    printf("Caught a memory exception.\n");
}
AND_CATCH_ALL (e) {
    printf("Caught an exception.\n");
}
END_CATCH_ALL
```

THROW宏相当于C++语言中的throw指令；你以什么类型做为THROW的参数，就会有一个相对应的 AfxThrow_ 函数被调用（这是台面下的行为）：

以下是MFC 4.x 的exceptions 宏定义：

MFC Exception Type	MFC Throw Function	DOS support	Windows support
CException		v	v
CMemoryException	AfxThrowMemoryException	v	v
CFileException	AfxThrowFileException	v	v
CArchiveException	AfxThrowArchiveException	v	v
CNotSupportedException	AfxThrowNotSupportedException	v	v
CResourceException	AfxThrowResourceException		v
COleException	AfxThrowOleException		v
COleDispatchException	AfxThrowOleDispatchException		v
CDBException	AfxThrowDBException		v
CDaoException	AfxThrowDaoException		v
CUserException	AfxThrowUserException		v

```
// in AFX.H
////////////////////////////////////////////////////////////////////
// Exception macros using try, catch and throw
//  (for backward compatibility to previous versions of MFC)
#ifndef _AFX_OLD_EXCEPTIONS
#define TRY { AFX_EXCEPTION_LINK _afxExceptionLink; try {
#define CATCH(class, e) } catch (class* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
_afxExceptionLink.m_pException = e;
#define AND_CATCH(class, e) } catch (class* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
_afxExceptionLink.m_pException = e;
#define END_CATCH } }
#define THROW(e) throw e
#define THROW_LAST() (AfxThrowLastCleanup(), throw)
// Advanced macros for smaller code
#define CATCH_ALL(e) } catch (CException* e) \
{ { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e;
#define AND_CATCH_ALL(e) } catch (CException* e) \
{ { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e;
#define END_CATCH_ALL } } }
#define END_TRY } catch (CException* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e; } }
#else //_AFX_OLD_EXCEPTIONS
////////////////////////////////////////////////////////////////////
// Exception macros using setjmp and longjmp
//  (for portability to compilers with no support for C++ exception handling)
#define TRY \
{ AFX_EXCEPTION_LINK _afxExceptionLink; \
if (::setjmp(_afxExceptionLink.m_jumpBuf) == 0)
#define CATCH(class, e) \
else if (::AfxCatchProc(RUNTIME_CLASS(class))) \
{ class* e = (class*)_afxExceptionLink.m_pException;
#define AND_CATCH(class, e) \
} else if (::AfxCatchProc(RUNTIME_CLASS(class))) \
{ class* e = (class*)_afxExceptionLink.m_pException;
#define END_CATCH \
} else { ::AfxThrow(NULL); } }
#define THROW(e) AfxThrow(e)
#define THROW_LAST() AfxThrow(NULL)
// Advanced macros for smaller code
#define CATCH_ALL(e) \
else { CException* e = _afxExceptionLink.m_pException;
#define AND_CATCH_ALL(e) \
} else { CException* e = _afxExceptionLink.m_pException;
#define END_CATCH_ALL } }
#define END_TRY }
#endif //_AFX_OLD_EXCEPTIONS
```

Template

这并不是一本C++书籍，我也并不打算介绍太多距离「运用 MFC」主题太远的C++论题。Template 虽然很重要，但它与「运用MFC」有什么关系？有！第 8 章当我们开始设计 Scribble 程序时，需要用到MFC 的collection classes，而这一组类自从MFC 3.0以来就有了 template 版本（因为Visual C++ 编译器从 2.0 版开始支持 C++ template）。运用之前，我们总该了解一下新的语法、精神、以及应用。

到底什么是template？重要性如何？Kaare Christian在1994/01/25 的 PC-Magazine上有一篇文章，说得很好：

无性生殖并不只是存在于遗传工程上，对程序员而言它也是一个由来已久的动作。过去，我们只不过是以一个简单而基本的工具，也就是一个文字编辑器，重制我们的程序代码。今天，C++ 提供给我们一个更好的繁殖方法：template。

复制一段既有程序代码的一个最平常的理由就是为了改变数据类型。举个例子，假设你写了一个绘图函数，使用整数 x, y 坐标；突然之间你需要相同的程序代码，但坐标值改采long。你当然可以使用一个文字编辑器把这段代码复制一份，然后把其中的数据类型改变过来。有了C++，你甚至可以使用重载（overloaded）函数，那么你就可以仍旧使用相同的函数名称。函数的重载的确使我们有比较清爽的程序代码，但它们意味着你还是必须在你的程序的许多地方维护完全相同的算法。

C语言对此问题的解答是：使用宏。虽然你因此对于相同的算法只需写一次程序代码，但宏有它自己的缺点。第一，它只适用于简单的功能。第二个缺点比较严重：宏不提供数据类型检验，因此牺牲了 C++ 的一个主要效益。第三个缺点是：宏并非函数，程序中任何调用宏的地方都会被编译器前置处理器原原本本地插入宏所定义的那一段代码，而非只是一个函数调用，因此你每使用一次宏，你的执行文件就会膨胀一点。

Templates 提供比较好的解决方案，它把「一般性的算法」和其「对数据类型的实现部分」区分开来。你可以先写算法的程序代码，稍后在使用时再填入实际数据类型。新的 C++ 语法使「数据类型」也以参数的姿态出现。有了 template，你可以拥有宏「只写一次」的优点，以及重载函数「类型检验」的优点。

C++ 的template 有两种，一种针对function，另一种针对class。

Template Functions

假设我们需要一个计算数值幂次方的函数，名曰power。我们只接受正幂次方数，如果是负幂次方，就让结果为0。

对于整数，我们的函数应该是这样：

```
#0001  int power(int base, int exponent)
#0002  {
#0003      int result = base;
#0004      if (exponent == 0)  return (int)1;
#0005      if (exponent < 0)   return (int)0;
#0006      while (--exponent)  result *= base;
#0007      return result;
#0008  }
```

对于长整数，函数应该是这样：

```
#0001  long power(long base, int exponent)
#0002  {
#0003      long result = base;
#0004      if (exponent == 0)  return (long)1;
#0005      if (exponent < 0)   return (long)0;
#0006      while (--exponent)  result *= base;
#0007      return result;
#0008  }
```

对于浮点数，我们应该...，对于复数，我们应该....。喔喔，为什么不能够把数据类型也变成参数之一，在使用时指定呢？是的，这就是 template 的妙用：

```
template <class T> T power(T base, int exponent);
```

写成两行或许比较清楚：

```
template <class T>
T power(T base, int exponent);
```

这样的函数声明是以一个特殊的template前缀开始，后面紧跟着一个参数列（本例只一个参数）。容易让人迷惑的是其中的 "class" 字眼，它其实并不一定表示C++的class，它也可以是一个普通的数据类型。 `<class T>` 只不过是表示：T 是一种类型，而此一类型将在调用此函数时才给予。

下面就是power函数的template版本：

```
#0001  template <class T>
#0002  T power(T base, int exponent)
#0003  {
#0004      T result = base;
#0005      if (exponent == 0)  return (T)1;
#0006      if (exponent < 0)   return (T)0;
#0007      while (--exponent)  result *= base;
#0008      return result;
#0009  }
```

传回值必须确保为类型T，以吻合template函数的声明。

下面是template函数的调用方法：

```
#0001  #include <iostream.h>
#0002  void main()
#0003  {
#0004      int i = power(5, 4);
#0005      long l = power(1000L, 3);
#0006      long double d = power((long double)1e5, 2);
#0008      cout << "i= " << i << endl;
#0009      cout << "l= " << l << endl;
#0010      cout << "d= " << d << endl;
#0011  }
```

执行结果如下：

```
i= 625
l= 1000000000
d= 1e+010
```

在第一次调用中，T变成int，在第二次调用中，T变成long。而在第三次调用中，T 又成为了一个 long double。但如果调用时候把数据类型混乱掉了，像这样：

```
int i = power(1000L, 4);  // 基值是个 long，传回值却是个 int。错误示范！
```

编译时就会出错。

template 函数的数据类型参数 T 究竟可以适应多少种类型？我要说，几乎「任何数据类型」都可以，但函数中对该类型数值的任何运算动作，都必须支持——否则编译器就不知道该怎么办了。以 power 函数为例，它对于 result 和 base 两个数值的运算动作有：

```
1.  T result = base;
2.  return (T)1;
3.  return (T)0;
4.  result *= base;
5.  return result;
```

C++ 所有内建数据类型如int或long都支持上述运算动作。但如果你为某个 C++ 类产生一个 power 函数，那么这个C++类必须包含适当的成员函数以支持上述动作。

如果你打算在template函数中以C++ 类代替class T，你必须清楚知道哪些运算动作曾被使用于此一函数中，然后在你的C++类中把它们全部实现出来。否则，出现的错误耐人寻味。

Template Classes

我们也可以建立template classes，使它们能够神奇地操作任何类型的数据。下面这个例子是让CThree 类储存三个成员变量，成员函数Min传回其中的最小值，成员函数Max则传回其中的最大值。我们把它设计为template class，以便这个类能适用于各式各样的数据类型：

```
#0001  template <class T>
#0002  class CThree
#0003  {
#0004   public :
#0005       CThree(T t1, T t2, T t3);
#0006       T Min();
#0007       T Max();
#0008   private:
#0009       T a, b, c;
#0010  };
```

语法还不至于太稀奇古怪，把T看成是大家熟悉的int或float也就是了。下面是成员函数的定义：

```
#0001  template <class T>
#0002  T CThree<T>::Min()
#0003  {
#0004      T minab = a < b ? a : b;
#0005      return minab < c ? minab : c;
#0006  }
#0008  template <class T>
#0009  T CThree<T>::Max()
#0010  {
#0011      T maxab = a < b ? b : a;
#0012      return maxab < c ? c : maxab;
#0013  }
#0015  te mplate <class T>
#0016  CThree <T>::CThree(T t1, T t2, T t3) :
#0017  a(t1), b(t2), c(t3)
#0018  {
#0019      return;
#0020  }
```

这里就得多注意些了。每一个成员函数前都要加上`template <class T>`，而且类名称应该使用`CThree<T>`。

以下是 template class 的使用方式：

```
#0001  #include <iostream.h>
#0002  void main()
#0003  {
#0004      CThree<int> obj1(2, 5, 4);
#0005      cout << obj1.Min() << endl;
#0006      cout << obj1.Max() << endl;
#0008      CThree<float> obj2(8.52, -6.75, 4.54);
#0009      cout << obj2.Min() << endl;
#0010      cout << obj2.Max() << endl;
#0012      CThree<long> obj3(646600L, 437847L, 364873L);
#0013      cout << obj3.Min() << endl;
#0014      cout << obj3.Max() << endl;
#0015  }
```

执行结果如下：

```
2
5
-6.75
8.52
364873
646600
```

稍早我曾说过，只有当 template 函数对于数据类型 T支持所有必要的运算动作时，T 才得被视为有效。此一限制对于template classes亦属实。为了针对某些类产生一个CThree，该类必须提供copy构造函数以及operator<，因为它们是Min和Max成员函数中对 T 的运算动作。

但是如果你用的是别人template classes，你又如何知道什么样的运算动作是必须的呢？唔，该template classes的说明文件中应该有所说明。如果没有，只有原始代码才能揭露秘密。C++ 内建数据类型如int和float等不需要在意这份要求，因为所有内建的数据型别都支持所有的标准运算动作。

Templates 的编译与链接

对程序员而言C++ templates可说是十分容易设计与使用，但对于编译器和链接器而言却是一大挑战。编译器遇到一个template时，不能够立刻为它产生机器代码，它必须等待，直到template被指定某种类型。从程序员的观点来看，这意味着template function或template class的完整定义将出现在 template 被使用的每一个角落，否则，编译器就没有足够的信息可以帮助产生目的代码。当多个源文件使用同一个 template 时，事情更趋复杂。

随着编译器的不同，掌握这种复杂度的技术也不同。有一个常用的技术，Borland 称之为Smart，应该算是最容易的：每一个使用Template 的程序代码的目的文件中都存在有template代码，链接器负责复制和删除。

假设我们有一个程序，包含两个源文件A.CPP 和B.CPP，以及一个THREE.H（其内定义了一个template类，名为CThree）。A.CPP 和 B.CPP 都含入 THREE.H。如果 A.CPP以 int 和double 使用这个template 类，编译器将在 A.OBJ 中产生 int 和 double 两种版本的template类可执行代码。如果 B.CPP 以 int 和 float 使用这个 template 类，编译器将在 B.OBJ中产生int和 float 两种版本的template类可执行代码。即使虽然A.OBJ中已经有一个 int 版了，编译器没有办法知道。

然后，在链接过程中，所有重复的部分将被删除。请看图 2-1。

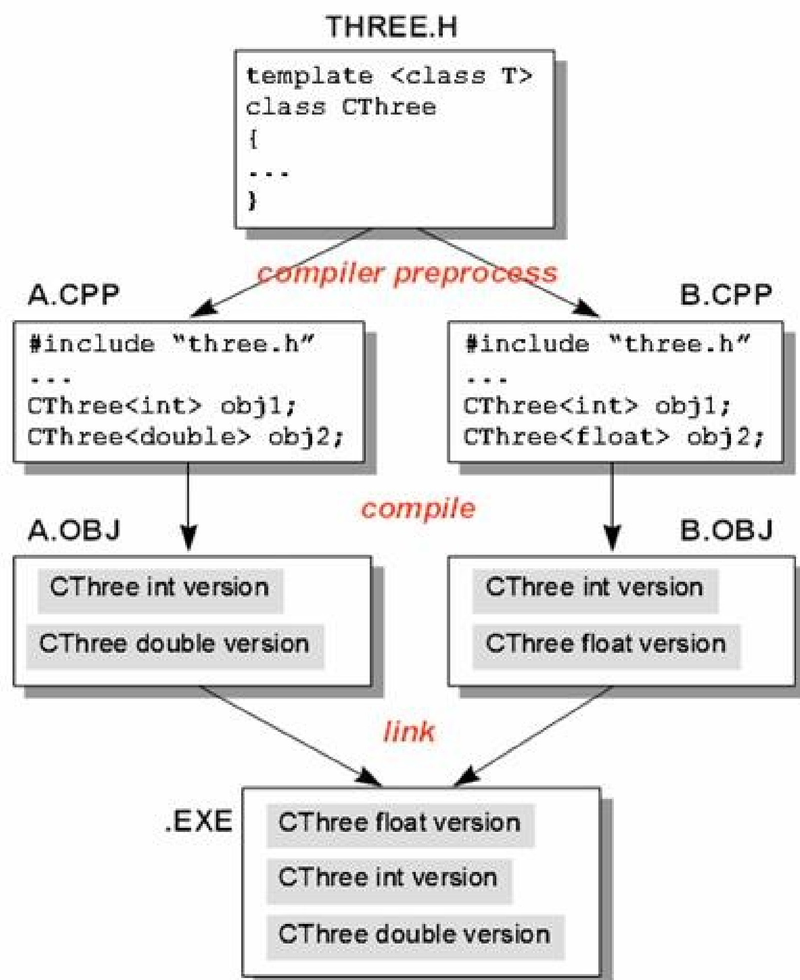


图2-1 链接器会把所有赘余的template 代码剔除。这在Borland 链接器里

头称为smart技术。其它链接器亦使用类似的技术。

第 3 章 MFC六大关键技术之模拟

> 演化（evolution）永远在进行，
>
> 这个世界却不是每天都有革命（revolution）发生。

Application Framework 在软件界确实称得上具有革命精神。

整个 MFC 4.0 多达 189 个类，原始代码达 252 个实现文件，58 个头文件，共 10 MB 之多。MFC 4.2 又多加了 29 个类。这么庞大的对象，当然不是每一个类每一个数据结构都是我的仿真目标。我只挑选最神秘又最重要，与应用程序主干息息相关的题目，包括：

- MFC 程序的初始化过程
- RTTI（Runtime Type Information）运行时类型信息
- Dynamic Creation 动态生成
- Persistence 永续留存
- Message Mapping 消息映射
- Message Routing 消息循环

MFC本身的设计在Application Framework之中不见得最好，敌视者甚至认为它是个Minotaur（注）！但无论如何，这是当今软件霸主微软公司的产品，从探究application framework 设计的角度来说，实为一个重要参考；而如果从选择一套 application framework 作为软件开发工具的角度来说，单就就业市场的需求，我对 MFC 的推荐再加 10 分！

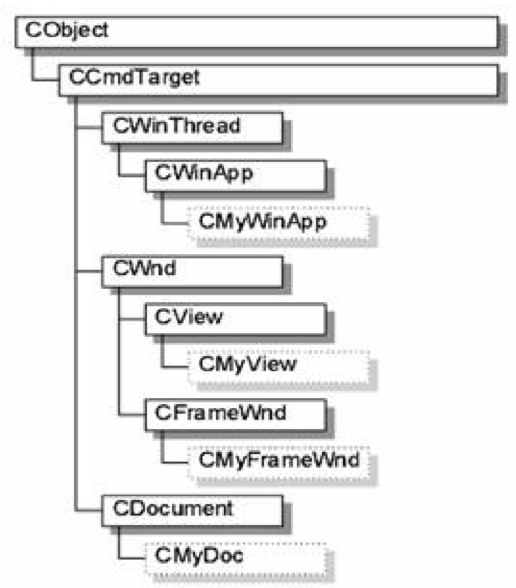
注：Minotaur 是希腊神话中的牛头人身怪物，居住在迷宫之中。进入迷宫的人如果走不出来，就会被牠一口吃掉。

以下所有程序的类阶层架构、类名称、变量名称、结构名称、函数名称、函数行为，都以MFC为仿真对象，具体而微。也可以说，我从数以万行计的MFC原始代码中，「偷」了一些出来，砍掉旁枝末节，只露出重点。

在文件的安排上，我把模拟MFC的类都集中在MFC.H 和MFC.CPP中，把自己派生的类集中在MY.H 和MY.CPP 中。对于自定类，我的命名方式是在父类的名称前面加一个 "My"，例如派生自CWinApp 者，名为CMyWinApp，派生自 CDocument者，名为 CMyDoc。

MFC 类阶层

首先我以一个极简单的程序 Frame1，把 MFC 数个最重要类的阶层关系模拟出来：




```
这个实例仿真MFC的类阶层。后续数节中，我会继续在这个类阶层上开发新的能力。在这些名为Frame? 的各范例中，我
```

Frame1 范例程序

MFC.H

```
#0001  #include <iostream.h>
#0003  class CObject
#0004  {
#0005   public:
#0006       CObject::CObject()  {  cout << "CObject Constructor \n";  }
#0007       CObject::~CObject() {  cout << "CObject Destructor \n";   }
#0008  };
#0010  class CCmdTarget : public CObject
#0011  {
#0012   public:
#0013   CCmdTarget::CCmdTarget(){cout << "CCmdTarget Constructor \n"; }
#0014   CCmdTarget::~CCmdTarget(){cout << "CCmdTarget Destructor\n";}
#0015  };
#0017  class CWinThread : public CCmdTarget
#0018  {
#0019   public:
#0020   CWinThread::CWinThread(){cout<<"CWinThread Constructor\n"; }
#0021   CWinThread::~CWinThread(){cout<<"CWinThread Destructor\n";  }
#0022  };
#0024  class CWinApp : public CWinThread
#0025  {
#0026   public:
#0027       CWinApp* m_pCurrentWinApp;
#0029   public:
#0030       CWinApp::CWinApp()  { m_pCurrentWinApp = this;
                             cout << "CWinApp Constructor \n";  }
#0031       CWinApp::~CWinApp() { cout << "CWinApp Destructor \n";   }
#0032  };
#0034  class CDocument : public CCmdTarget
#0035  {
#0036   public:
#0037       CDocument::CDocument(){cout<< "CDocument Constructor \n"; }
#0038       CDocument::~CDocument(){cout<<"CDocument Destructor \n";}
#0039  };
#0042  class CWnd : public CCmdTarget
#0043  {
#0044   public:
#0045       CWnd::CWnd(){cout << "CWnd Constructor \n"; }
#0046       CWnd::~CWnd(){cout<< "CWnd Destructor \n";  }
#0047  };
#0049  class CFrameWnd : public CWnd
#0050  {
#0051   public:
#0052       CFrameWnd::CFrameWnd(){cout<< "CFrameWnd Constructor \n"; }
#0053       CFrameWnd::~CFrameWnd(){cout<<"CFrameWnd Destructor \n";}
#0054  };
#0056  class CView : public CWnd
#0057  {
#0058   public:
#0059       CView::CView(){  cout << "CView Constructor \n"; }
#0060       CView::~CView()  {  cout << "CView Destructor \n";  }
#0061  };
#0064  // global function
#0066  CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001  #include "my.h"//原本含入mfc.h就好，但为了CMyWinApp的定义，所以...
#0003  extern CMyWinApp theApp;
#0005  CWinApp* AfxGetApp()
#0006  {
#0007      return theApp.m_pCurrentWinApp;
#0008  }
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006   public:
#0007      CMyWinApp::CMyWinApp(){ cout<< "CMyWinApp Constructor \n"; }
#0008      CMyWinApp::~CMyWinApp(){cout<< "CMyWinApp Destructor \n";}
#0009  };
#0010
#0011  class CMyFrameWnd : public CFrameWnd
#0012  {
#0013   public:
#0014       CMyFrameWnd(){  cout << "CMyFrameWnd Constructor \n";}
#0015       ~CMyFrameWnd(){ cout << "CMyFrameWnd Destructor \n";}
#0016  };
```

MY.CPP

```
#0001  #include "my.h"
#0002
#0003  CMyWinApp theApp;  // global object
#0006  // main
#0008  void main()
#0009  {
#0011      CWinApp* pApp = AfxGetApp();
#0013  }
```

Frame1 的命令列编译链接动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

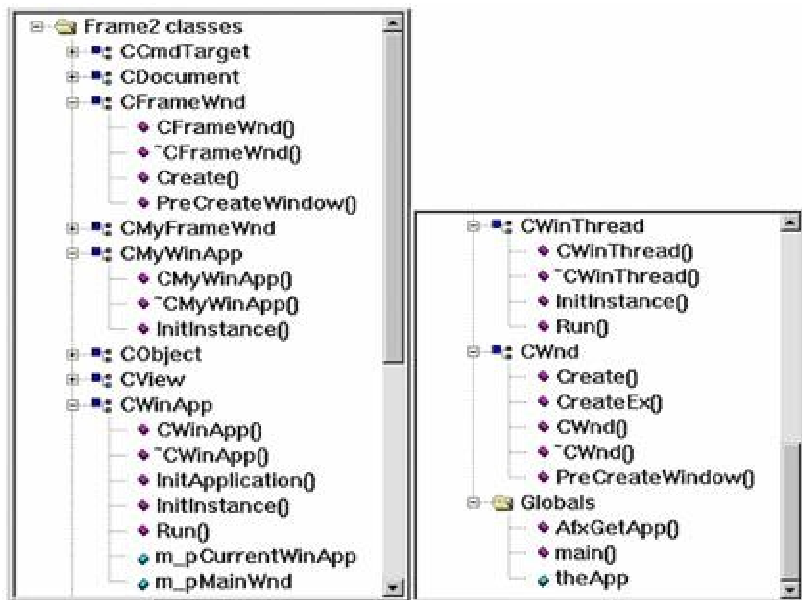
Frame1 的执行结果是：

```
CObject Constructor
CCmdTarget Constructor
CWinThread Constructor
CWinApp Constructor
CMyWinApp Constructor
CMyWinApp Destructor
CWinApp Destructor
CWinThread Destructor
CCmdTarget Destructor
CObject Destructor
```

好，你看到了，Frame1 并没有new 任何对象，反倒是有一个全局对象theApp 存在。C++ 规定，全局对象的构造将比程序进入点（在 DOS 环境为main，在 Windows 环境为WinMain）更早。所以theApp 的构造函数将更早于main。换句话说你所看到的执行结果中的那些构造函数输出动作全都是在main函数之前完成的。main 函数调用全局函数 AfxGetApp 以取得 theApp 的对象指针。这完全是仿真 MFC 程序的手法。

MFC 程序的初始化过程

MFC程序也是个Windows程序，它的内部一定也像第 1 章所述一样，有窗口注册动作，有窗口产生动作，有消息循环动作，也有窗口函数。此刻我并不打算做出 Windows 程序，只是想交待给你一个程序流程，这个流程正是任何 MFC 程序的初始化过程的简化。



就如我曾在第 1 章解释过的，InitApplication 和InitInstance 现在成了MFC的CWinApp的两个虚函数。前者负责「每一个程序只做一次」的动作，后者负责「每一个执行个体都得做一次」的动作。通常，系统会（并且有能力）为你注册一些标准的窗口类（当然也就准备好了一些标准的窗口函数），你（应用程序设计者）应该在你的CMyWinApp中改写InitInstance，并在其中把窗口产生出来 -- 这样你才有机会在标准的窗口类中指定自己的窗口标题和菜单。下面就是我们新的main 函数：

```
// MY.CPP
CMyWinApp theApp;
void main()
{
CWinApp* pApp = AfxGetApp();
pApp->InitApplication();
pApp->InitInstance();
pApp->Run();
}
```

其中pApp指向theApp全局对象。在这里我们开始看到了虚函数的妙用（还不熟练者请快复习第 2 章）：

pApp->InitApplication()调用的是CWinApp::InitApplication， pApp->InitInstance()调用的是CMyWinApp::InitInstance（因为CMyWinApp改写它了），pApp->Run()调用的是CWinApp::Run，好，请注意以下 CMyWinApp::InitInstance 的动作，以及它所引发的行为：

```
BOOL CMyWinApp::InitInstance()
{
    cout << "CMyWinApp::InitInstance \n";
    m_pMainWnd = new CMyFrameWnd;  // 引发 CMyFrameWnd::CMyFrameWnd 构造函数
    return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
    Create();  // Create 是虚函数，但 CMyFrameWnd 未改写它，所以引发父类的
               // CFrameWnd::Create
}

BOOL CFrameWnd::Create()
{

  cout << "CFrameWnd::Create \n";
  CreateEx();  // CreateEx 是虚函数，但 CFrameWnd 未改写之，所以引发
               // CWnd::CreateEx
  return TRUE;
}

BOOL CWnd::CreateEx()
{
  cout << "CWnd::CreateEx \n";
  PreCreateWindow();     // 这是一个虚函数，CWnd 中有定义，CFrameWnd 也改写了
                         // 它。那么你说这里到底是调用 CWnd::PreCreateWindow 还是
                         // CFrameWnd::PreCreateWindow 呢?
  return TRUE;
}

                  答案是 CFrameWnd::PreCreateWindow。
                  这便是我在第 2 章的「Object slicing 与虚函数」
                  一节所说提「虚函数的一个极重要的行为模式」。

BOOL CFrameWnd::PreCreateWindow()
{
  cout << "CFrameWnd::PreCreateWindow \n";
  return TRUE;
}
```

你看到了，这些函数什么正经事儿也没做，光只输出一个标识符串。我主要的目的是在让你先熟悉MFC程序的执行流程。

Frame2 的命令列编译链接动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

以下就是 Frame2 的执行结果：

```
CWinApp::InitApplication
CMyWinApp::InitInstance
CMyFrameWnd::CMyFrameWnd
CFrameWnd::Create
CWnd::CreateEx
CFrameWnd::PreCreateWindow
CWinApp::Run
CWinThread::Run
```

Frame2 范例程序

MFC.H

```
#0001  #define BOOL int
#0002  #define TRUE 1
#0003  #define FALSE 0
#0004
#0005  #include <iostream.h>
#0006
#0007  class CObject
#0008  {
#0009  public:
#0010      CObject::CObject()  { }
#0011      CObject::~CObject() { }
#0012  };
#0013
#0014  class CCmdTarget : public CObject
#0015  {
#0016   public:
#0017       CCmdTarget::CCmdTarget()  {  }
#0018       CCmdTarget::~CCmdTarget() {  }
#0019  };
#0020
#0021  class CWinThread : public CCmdTarget
#0022  {
#0023  public:
#0024  CWinThread::CWinThread()  {  }
#0025  CWinThread::~CWinThread() {  }
#0026
#0027  virtual BOOL InitInstance() {
#0028                           cout << "CWinThread::InitInstance \n";
#0029                           return TRUE;
#0030                           }
#0031  virtual int Run() {
#0032                     cout << "CWinThread::Run \n";
#0033                     return 1;
#0034                     }
#0035  };
#0036
#0037  class CWnd;
#0038
#0039  class CWinApp : public CWinThread
#0040  {
#0041  public:
#0042      CWinApp* m_pCurrentWinApp;
#0043      CWnd* m_pMainWnd;
#0044
#0045  public:
#0046      CWinApp::CWinApp()  { m_pCurrentWinApp = this; }
#0047      CWinApp::~CWinApp() {   }
#0048
#0049  virtual BOOL InitApplication() {
#0050                         cout << "CWinApp::InitApplication \n";
#0051                         return TRUE;
#0052                         }
```

```
#0053  virtual BOOL InitInstance(){
#0054                       cout << "CWinApp::InitInstance \n";
#0055                       return TRUE;
#0056                       }
#0057  virtual int Run() {
#0058                     cout << "CWinApp::Run \n";
#0059                     return CWinThread::Run();
#0060                    }
#0061  };
#0062
#0063
#0064  class CDocument : public CCmdTarget
#0065  {
#0066  public:
#0067      CDocument::CDocument(){  }
#0068      CDocument::~CDocument()  {  }
#0069  };
#0070
#0071
#0072  class CWnd : public CCmdTarget
#0073  {
#0074  public:
#0075      CWnd::CWnd(){  }
#0076      CWnd::~CWnd() {  }
#0077
#0078  virtual BOOL Create();
#0079  BOOL CreateEx();
#0080  virtual BOOL PreCreateWindow();
#0081  };
#0082
#0083  class CFrameWnd : public CWnd
#0084  {
#0085  public:
#0086      CFrameWnd::CFrameWnd(){  }
#0087      CFrameWnd::~CFrameWnd() {  }
#0088  BOOL Create();
#0089  virtual BOOL PreCreateWindow();
#0090  };
#0091
#0092  class CView : public CWnd
#0093  {
#0094  public:
#0095      CView::CView(){  }
#0096      CView::~CView() {  }
#0097  };
#0098
#0099
#0100  // global function
#0101  CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001  #include "my.h"  // 原该含入 mfc.h 就好，但为了 CMyWinApp 的定义，所以...
#0002
#0003  extern CMyWinApp theApp;  // external global object
#0004
#0005  BOOL CWnd::Create()
#0006  {
#0007     cout << "CWnd::Create \n";
#0008     return TRUE;
#0009  }
#0010
#0011  BOOL CWnd::CreateEx()
#0012  {
#0013     cout << "CWnd::CreateEx \n";
#0014     PreCreateWindow();
#0015     return TRUE;
#0016  }
#0017
#0018  BOOL CWnd::PreCreateWindow()
#0019  {
#0020     cout << "CWnd::PreCreateWindow \n";
#0021     return TRUE;
#0022  }
#0023
#0024  BOOL CFrameWnd::Create()
#0025  {
#0026     cout << "CFrameWnd::Create \n";
#0027     CreateEx();
#0028     return TRUE;
#0029  }
#0030
#0031  BOOL CFrameWnd::PreCreateWindow()
#0032  {
#0033     cout << "CFrameWnd::PreCreateWindow \n";
#0034     return TRUE;
#0035  }
#0036
#0037
#0038  CWinApp* AfxGetApp()
#0039  {
#0040     return theApp.m_pCurrentWinApp;
#0041  }
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007     CMyWinApp::CMyWinApp()      {    }
#0008     CMyWinApp::~CMyWinApp()   {      }
#0009
#0010     virtual BOOL InitInstance();
#0011  };
#0012
#0013  class CMyFrameWnd : public CFrameWnd
#0014  {
#0015  public:
#0016     CMyFrameWnd();
#0017     ~CMyFrameWnd()  {      }
#0018  };
```

MY.CPP

```
#0001  #include "my.h"
#0002
#0003  CMyWinApp theApp;  // global object
#0004
#0005  BOOL CMyWinApp::InitInstance()
#0006  {
#0007      cout << "CMyWinApp::InitInstance \n";
#0008      m_pMainWnd = new CMyFrameWnd;
#0009      return TRUE;
#0010  }
#0011
#0012  CMyFrameWnd::CMyFrameWnd()
#0013  {
#0014      cout << "CMyFrameWnd::CMyFrameWnd \n";
#0015      Create();
#0016  }
#0017
#0018  //------------------------------------------------------------------
#0019  // main
#0020  //------------------------------------------------------------------
#0021  void main()
#0022  {
#0023
#0024      CWinApp* pApp = AfxGetApp();
#0025
#0026      pApp->InitApplication();
#0027      pApp->InitInstance();
#0028      pApp->Run();
#0029  }
```

RTTI（运行时类型辨识）

你已经在第 2 章看到，Visual C++ 4.0 支持RTTI，重点不外乎是：

1. 编译时需选用/GR 选项（/GR 的意思是enable C++ RTTI）

2. 含入typeinfo.h

3. 使用新的typeid 运算符。

RTTI 亦有称为 Runtime Type Identification 者。

MFC 早在编译器支持 RTTI 之前，就有了这项能力。我们现在要以相同的手法，在DOS程序中仿真出来。我希望我的类库具备IsKindOf的能力，能在运行时侦测某个对象是否「属于某种类」，并传回 TRUE 或 FALSE。以前一章的 Shape 为例，我希望：

```
CSquare* pSquare = new CSquare;
cout << pSquare->IsKindOf(CSquare); //应该获得 1 (TRUE)
cout << pSquare->IsKindOf(CRect); // 应该获得 1 (TRUE)
cout << pSquare->IsKindOf(CShape); // 应该获得 1 (TRUE)
```

```
cout << pSquare->IsKindOf(CCircle);       // 应该获得 0 (FALSE)

以 MFC 的类阶层来说，我希望：                注意：真正的 IsKindOf 参数其实没能那么单纯

CMyDoc* pMyDoc = new CMyDoc;
cout << pMyDoc->IsKindOf(CMyDoc);         // 应该获得 1 (TRUE)
cout << pMyDoc->IsKindOf(CDocument);      // 应该获得 1 (TRUE)
cout << pMyDoc->IsKindOf(CCmdTarget); // 应该获得 1 (TRUE)
cout << pMyDoc->IsKindOf(CWnd);       // 应该获得 0 (FALSE)
```

类型录网与 CRuntimeClass

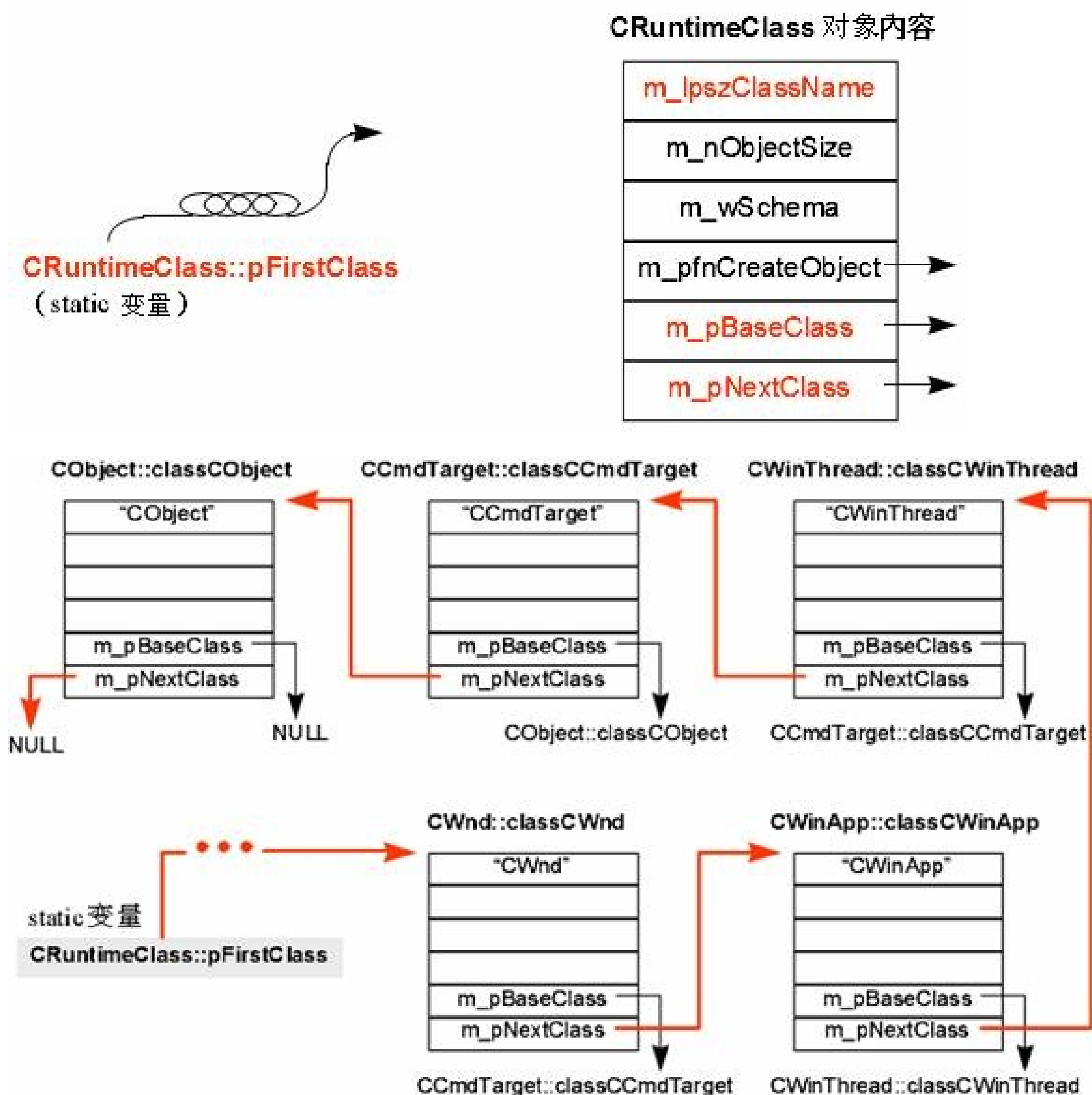
怎么设计RTTI呢？让我们想想，当你手上握有一种色泽，想知道它的RGB成份比，不查色表行吗？当你持有一种产品，想知道它的型号，不查型录行吗？要达到RTTI 的能力，我们（类库的设计者）一定要在类构造起来的时候，记录必要的信息，以建立型录。型录中的类信息，最好以串列（linked list）方式串接起来，将来方便一一比对。

我们这份「类型录」的串列元素将以CRuntimeClass描述之，那是一个结构，内中至少需有类名称、串列的 Next 指针，以及串列的 First 指针。由于 First 指针属于全局变量，一份就好，所以它应该以 static 修饰之。除此之外你所看到的其它 CRuntimeClass成员都是为了其它目的而准备，陆陆续续我会介绍出来。

```
// in MFC.H
struct CRuntimeClass
{
// Attributes
        LPCSTR m_lpszClassName;
        int m_nObjectSize;
        UINT m_wSchema; // schema number of the loaded class
        CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
        CRuntimeClass* m_pBaseClass;

        // CRuntimeClass objects linked together in simple list
        static CRuntimeClass* pFirstClass; // start of class list
        CRuntimeClass* m_pNextClass;       // linked list of registered classes
};
```

我希望，每一个类都能拥有这样一个 CRuntimeClass 成员变量，并且最好有一定的命名规则（例如在类名称之前冠以 "class" 作为它的名称），然后，经由某种手段将整个类库构造好之后，「类型录」能呈现类似这样的风貌：



DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC 宏

为了神不知鬼不觉把 CRuntimeClass 对象塞到类之中，并声明一个可以抓到该对象位址的函数，我们定义 DECLARE_DYNAMIC 宏如下：

```
#define DECLARE_DYNAMIC(class name) \
public: \
        static CRuntimeClass class##class_name; \
        virtual CRuntimeClass* GetRuntimeClass() const;
```

出现在宏定义之中的 ##，用来告诉编译器，把两个字符串系在一起。如果你这么使用此宏：

```
DECLARE_DYNAMIC(CView)
```

编译器前置处理器为你做出的代码是：

```
public:
        static CRuntimeClass classCView;
        virtual CRuntimeClass* GetRuntimeClass() const;
```

这下子，只要在声明类时放入 DECLARE_DYNAMIC宏即万事OK喽。

不，还没有 OK，类型录（也就是各个 CRuntimeClass 对象）的内容指定以及串接工作最好也能够神不知鬼不觉，我们于是再定义 IMPLEMENT_DYNAMIC 宏：

```
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
        _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
```

其中的 _IMPLEMENT_RUNTIMECLASS 又是一个宏。这样区分是为了此一宏在「动态生成」（下一节主题）时还会用到。

```
#define  _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name,wSchema,pfnNew) \
        static char _lpsz##class_name[] = #class_name; \
        CRuntimeClass class_name::class##class_name = { \
                _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
                        RUNTIME_CLASS(base_class_name), NULL }; \
        static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
        CRuntimeClass* class_name::GetRuntimeClass() const \
                { return &class_name::class##class_name; } \
```

其中又有 RUNTIME_CLASS 宏，定义如下：

```
#define RUNTIME_CLASS(class_name) \
        (&class_name::class##class_name)
```

看起来整个 IMPLEMENT_DYNAMIC 内容好像只是指定初值，不然，其曼妙处在于它所使用的一个 struct AFX_CLASSINIT，定义如下：

```
struct AFX_CLASSINIT
        { AFX_CLASSINIT(CRuntimeClass* pNewClass); };
```

这表示它有一个构造函数（别惊讶，C++ 的 struct 与 class 都有构造函数），定义如下：

```
AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
{
        pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
        CRuntimeClass::pFirstClass = pNewClass;
}
```

很明显，此构造函数负责linked list的串接工作。整组宏看起来有点吓人，其实也没有什么，文字代换而已。现在看看这个实例：

```
// in header file
class CView : public CWnd
{
        DECLARE_DYNAMIC(CView)
        ...
};

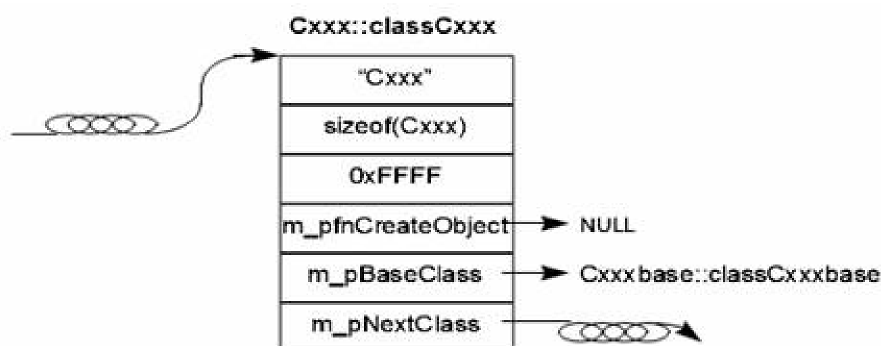
// in implementation file
IMPLEMENT_DYNAMIC(CView, CWnd)
```

上述的代码展开来成为：

```
// in header file
class CView : public CWnd
{
public:
        static CRuntimeClass classCView; \
        virtual CRuntimeClass* GetRuntimeClass() const;
        ...
};
```

```
// in implementation file
static char _lpszCView[] = "CView";
CRuntimeClass CView::classCView = {
        _lpszCView, sizeof(CView), 0xFFFF, NULL,
                &CWnd::classCWnd, NULL };
static AFX_CLASSINIT _init_CView(&CView::classCView);
CRuntimeClass* CView::GetRuntimeClass() const
        { return &CView::classCView; }
```

于是乎，程序中只需要简简单单的两个宏DECLARE_DYNAMIC(Cxxx)和IMPLEMENT_DYNAMIC(Cxxx, Cxxxbase)，就完成了构造数据并加入串列的工作：



可是你知道，串列的头，总是需要特别费心处理，不能够套用一般的串列行为模式。我们的类根源CObject，不能套用现成的宏DECLARE_DYNAMIC 和IMPLEMENT_DYNAMIC，必须特别设计如下：

```
// in header file
class CObject
{
public:
  virtual CRuntimeClass* GetRuntimeClass() const;
  ...
public:
  static CRuntimeClass classCObject;
};
```

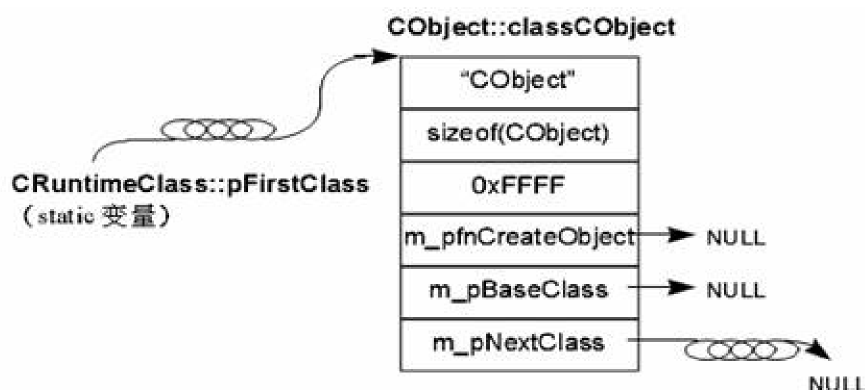
```
// in implementation file
static char szCObject[] = "CObject";
struct CRuntimeClass CObject::classCObject =
        { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
static AFX_CLASSINIT _init_CObject(&CObject::classCObject);

CRuntimeClass* CObject::GetRuntimeClass() const
{
        return &CObject::classCObject;
}
```

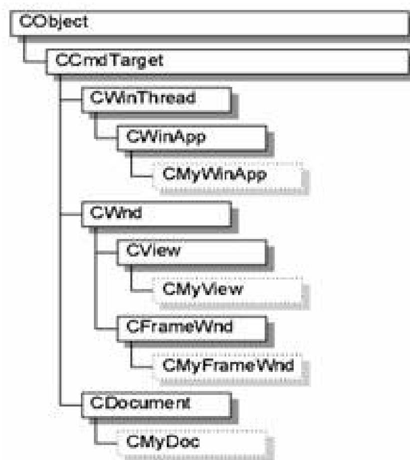
并且，CRuntimeClass 中的static 成员变量应该要初始化（如果你忘记了，赶快复习第 2 章的「静态成员（变量与函数）」一节）：

```
// in implementation file
CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
```

终于，整个「类型录」串列的头部就这样形成了：



范例程序Frame3在 .h文件中有这些类声明：




```
class CObject
{
...
};
class CCmdTarget : public CObject
{
        DECLARE_DYNAMIC(CCmdTarget)
...
};
class CWinThread : public CCmdTarget
{
        DECLARE_DYNAMIC(CWinThread)
...
};
class CWinApp : public CWinThread
{
        DECLARE_DYNAMIC(CWinApp)
...
};
class CDocument : public CCmdTarget
{
        DECLARE_DYNAMIC(CDocument)

...
};
class CWnd : public CCmdTarget
{
        DECLARE_DYNAMIC(CWnd)  // 其实在 MFC 中是 DECLARE_DYNCREATE()，见下节。
...
};
class CFrameWnd : public CWnd
{
      DECLARE_DYNAMIC(CFrameWnd) //其实在 MFC 中是 DECLARE_DYNCREATE()，见下节。
...
};
class CView : public CWnd
{
        DECLARE_DYNAMIC(CView)
...
};
class CMyWinApp : public CWinApp
{
...
};
class CMyFrameWnd : public CFrameWnd
{
...  // 其实在 MFC 应用程序中这里也有 DECLARE_DYNCREATE()，见下节。
};
class CMyDoc : public CDocument
{
```

```
...  // 其实在 MFC 应用程序中这里也有 DECLARE_DYNCREATE()，见下节。
};
class CMyView : public CView
{
...  // 其实在 MFC 应用程序中这里也有 DECLARE_DYNCREATE()，见下节。
};
```

范例程序Frame3在.cpp文件中有这些动作：

```
IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
IMPLEMENT_DYNAMIC(CWnd, CCmdTarget)  // 其实在 MFC 中它是 IMPLEMENT_DYNCREATE()，见下节。
IMPLEMENT_DYNAMIC(CFrameWnd, CWnd)   // 其实在 MFC 中它是 IMPLEMENT_DYNCREATE()，见下节
IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
IMPLEMENT_DYNAMIC(CView, CWnd)
```

于是组织出图 3-1 这样一个大网。

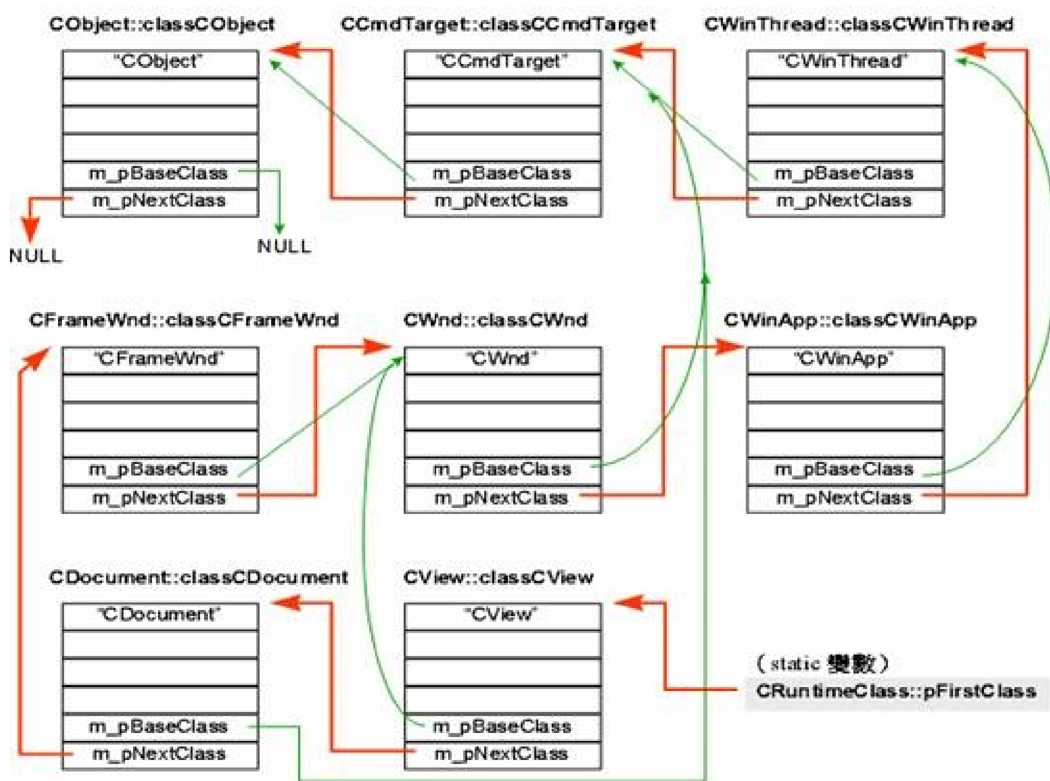


图3-1 CRuntimeClass 对象构成的类型录网。

本图只列出与RTTI 有关系的成员。

为了实证整个类型录网的存在，我在main 函数中调用PrintAllClasses，把串列中的每一个元素的类名称、对象大小、以及schema no. 印出来：

```
void PrintAllClasses()
{
    CRuntimeClass* pClass;

    // just walk through the simple list of registered classes
    for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
            pClass = pClass->m_pNextClass)
    {
        cout << pClass->m_lpszClassName << "\n";
        cout << pClass->m_nObjectSize << "\n";
        cout << pClass->m_wSchema << "\n";
    }
}
```

Frame3 的命令列编译链接动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

Frame3 的执行结果如下：

```
CView 4 65535 CDocument 4 65535 CFrameWnd 4 65535 CWnd 4 65535 CWinApp 12 65535 CWinThrea
```

Frame3 范例程序

MFC.H

```
#0001  #define BOOL int
#0002  #define TRUE 1
#0003  #define FALSE 0
```

```
#0004  #define LPCSTR  LPSTR
#0005  typedef char*   LPSTR;
#0006  #define UINT int
#0007  #define PASCAL _stdcall
#0008
#0009  #include <iostream.h>
#0010
#0011  class CObject;
#0012
#0013  struct CRuntimeClass
#0014  {
#0015  // Attributes
#0016          LPCSTR m_lpszClassName;
#0017          int m_nObjectSize;
#0018          UINT m_wSchema; // schema number of the loaded class
#0019          CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
#0020          CRuntimeClass* m_pBaseClass;
#0021
#0022          // CRuntimeClass objects linked together in simple list
#0023          static CRuntimeClass* pFirstClass; // start of class list
#0024          CRuntimeClass* m_pNextClass; // linked list of registered classes
#0025  };
#0026
#0027  struct AFX_CLASSINIT
#0028          { AFX_CLASSINIT(CRuntimeClass* pNewClass); };
#0029
#0030  #define RUNTIME_CLASS(class_name) \
#0031          (&class_name::class##class_name)
#0032
#0033  #define DECLARE_DYNAMIC(class_name) \
#0034  public: \
#0035          static CRuntimeClass class##class_name; \
#0036          virtual CRuntimeClass* GetRuntimeClass() const;
#0037
#0038  #define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
#0039          static char _lpsz##class_name[] = #class_name; \
#0040          CRuntimeClass class_name::class##class_name = { \
#0041                  _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
#0042                          RUNTIME_CLASS(base_class_name), NULL }; \
#0043          static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
#0044          CRuntimeClass* class_name::GetRuntimeClass() const \
#0045                  { return &class_name::class##class_name; } \
#0046
#0047  #define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
#0048          _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#0049
```

```
#0050  class CObject
#0051  {
#0052  public:
#0053    CObject::CObject()  {
#0054                        }
#0055    CObject::~CObject() {
#0056                        }
#0057
#0058    virtual CRuntimeClass* GetRuntimeClass() const;
#0059
#0060  public:
#0061    static CRuntimeClass classCObject;
#0062  };
#0063
#0064  class CCmdTarget : public CObject
#0065  {
#0066          DECLARE_DYNAMIC(CCmdTarget)
#0067  public:
#0068    CCmdTarget::CCmdTarget()  {
#0069                              }
#0070    CCmdTarget::~CCmdTarget() {
#0071                              }
#0072  };
#0073
#0074  class CWinThread : public CCmdTarget
#0075  {
#0076          DECLARE_DYNAMIC(CWinThread)
#0077  public:
#0078    CWinThread::CWinThread()  {
#0079                              }
#0080    CWinThread::~CWinThread() {
#0081                              }
#0082
#0083    virtual BOOL InitInstance() {
#0084                                  return TRUE;
#0085                                }
#0086    virtual int Run() {
#0087                        return 1;
#0088                      }
#0089  };
#0090
#0091  class CWnd;
#0092
#0093  class CWinApp : public CWinThread
#0094  {
#0095          DECLARE_DYNAMIC(CWinApp)
#0096  public:
#0097    CWinApp* m_pCurrentWinApp;
#0098    CWnd* m_pMainWnd;
#0099
#0100  public:
#0101    CWinApp::CWinApp()  {
#0102                          m_pCurrentWinApp = this;
#0103                        }
```

```
#0104 ·    CWinApp::~CWinApp() { ↵
#0105 ·                         } ↵
#0106 ·    ↵
#0107 ·    virtual BOOL InitApplication() { ↵
#0108 ·                                       return TRUE; ↵
#0109 ·                                       } ↵
#0110 ·    virtual BOOL InitInstance() ↵     {
#0111 ·                                     ↵return TRUE; ↵
#0112 ·    ↵                                 }
#0113 ·    virtual int Run() { ↵
#0114 ·                      return CWinThread::Run(); ↵
#0115 ·                      } ↵
#0116 }; ↵
#0117 ↵
#0118 class CDocument : public CCmdTarget ↵
#0119 { ↵
#0120        DECLARE_DYNAMIC(CDocument) ↵
#0121 public: ↵
#0122 ·    CDocument::CDocument() ↵    {
#0123 ·                                }
#0124 ·    CDocument::~CDocument()   { ↵
#0125 ·                                }
#0126 }; ↵
#0127 ↵
#0128 class CWnd : public CCmdTarget ↵
#0129 { ↵
#0130        DECLARE_DYNAMIC(CWnd) ↵
#0131 public: ↵
#0132 ·    CWnd::CWnd() ↵   {
#0133 ·                     }
#0134 ·    CWnd::~CWnd()  { ↵
#0135 ·                     } ↵
#0136 ·    ↵
#0137 ·    virtual BOOL Create(); ↵
#0138 ·    BOOL CreateEx(); ↵
#0139 ·    virtual BOOL PreCreateWindow(); ↵
#0140 }; ↵
#0141 ↵
```

```
#0142 class CFrameWnd : public CWnd
#0143 {
#0144        DECLARE_DYNAMIC(CFrameWnd)
#0145 public:
#0146   CFrameWnd::CFrameWnd()   {
#0147                            }
#0148   CFrameWnd::~CFrameWnd()  {
#0149                            }
#0150   BOOL Create();
#0151   virtual BOOL PreCreateWindow();
#0152 };
#0153
#0154 class CView : public CWnd
#0155 {
#0156        DECLARE_DYNAMIC(CView)
#0157 public:
#0158   CView::CView()   {
#0159                    }
#0160   CView::~CView()  {
#0161                    }
```

```
#0162  };
#0163
#0164
#0165  // global function
#0166  CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001  #include "my.h"  // 原该含入 mfc.h 就好，但为了 CMyWinApp 的定义，所以...
#0002
#0003  extern CMyWinApp theApp;
#0004
#0005  static char szCObject[] = "CObject";
#0006  struct CRuntimeClass CObject::classCObject =
#0007          { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
#0008  static AFX_CLASSINIT _init_CObject(&CObject::classCObject);
#0009
#0010  CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
#0011
#0012  AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
#0013  {
#0014          pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
#0015          CRuntimeClass::pFirstClass = pNewClass;
#0016  }
#0017
#0018  CRuntimeClass* CObject::GetRuntimeClass() const
#0019  {
#0020          return &CObject::classCObject;
#0021  }
#0022
#0023  BOOL CWnd::Create()
#0024  {
#0025     return TRUE;
#0026  }
#0027
#0028  BOOL CWnd::CreateEx()
#0029  {
#0030     PreCreateWindow();
#0031     return TRUE;
#0032  }
#0033
#0034  BOOL CWnd::PreCreateWindow()
#0035  {
#0036     return TRUE;
#0037  }
#0038
#0039  BOOL CFrameWnd::Create()
#0040  {
#0041     CreateEx();
#0042     return TRUE;
#0043  }
#0044
```

```
#0045  BOOL CFrameWnd::PreCreateWindow()
#0046  {
#0047      return TRUE;
#0048  }
#0049
```

```
#0050  IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
#0051  IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
#0052  IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
#0053  IMPLEMENT_DYNAMIC(CWnd, CCmdTarget)
#0054  IMPLEMENT_DYNAMIC(CFrameWnd, CWnd)
#0055  IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
#0056  IMPLEMENT_DYNAMIC(CView, CWnd)
#0057
#0058  // global function
#0059  CWinApp* AfxGetApp()
#0060  {
#0061      return theApp.m_pCurrentWinApp;
#0062  }
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007      CMyWinApp::CMyWinApp()      {
#0008                                  }
#0009      CMyWinApp::~CMyWinApp()   {
#0010                                  }
#0011
#0012      virtual BOOL InitInstance();
#0013  };
#0014
#0015  class CMyFrameWnd : public CFrameWnd
#0016  {
#0017  public:
#0018      CMyFrameWnd();
#0019      ~CMyFrameWnd()  {
#0020                        }
#0021  };
#0022
#0023
#0024  class CMyDoc : public CDocument
#0025  {
#0026  public:
#0027      CMyDoc::CMyDoc()  {
```

```
#0028                              }
#0029     CMyDoc::~CMyDoc()  {
#0030                              }
#0031  };
#0032
#0033  class CMyView : public CView
#0034  {
#0035  public:
#0036     CMyView::CMyView()       {
#0037                                }
#0038     CMyView::~CMyView()   {
#0039                                }
#0040  };
#0041
#0042  // global function
#0043  void PrintAllClasses();
```

MY.CPP

```
#0001  #include "my.h"
#0002
#0003  CMyWinApp theApp;
#0004
#0005  BOOL CMyWinApp::InitInstance()
#0006  {
#0007      m_pMainWnd = new CMyFrameWnd;
#0008      return TRUE;
#0009  }
#0010
#0011  CMyFrameWnd::CMyFrameWnd()
#0012  {
#0013      Create();
#0014  }
#0015
#0016  void PrintAllClasses()
#0017  {
#0018      CRuntimeClass* pClass;
#0019
#0020      // just walk through the simple list of registered classes
#0021      for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
#0022              pClass = pClass->m_pNextClass)
#0023      {
#0024          cout << pClass->m_lpszClassName << "\n";
#0025          cout << pClass->m_nObjectSize << "\n";
#0026          cout << pClass->m_wSchema << "\n";
#0027      }
#0028  }
#0029  //------------------------------------------------------------------
#0030  // main
#0031  //------------------------------------------------------------------
#0032  void main()
#0033  {
#0034    CWinApp* pApp = AfxGetApp();
#0035
#0036    pApp->InitApplication();
#0037    pApp->InitInstance();
#0038    pApp->Run();
#0039
#0040    PrintAllClasses();
#0041  }
```

IsKindOf（类型辨识）

有了图 3-1 这张「类型录」网，要实现IsKindOf 功能，再轻松不过了：

1. 为CObject加上一个IsKindOf 函数，于是此函数将被所有类继承。它将把参数所指定的某个CRuntimeClass对象拿来与类型录中的元素一一比对。比对成功（在型录中有发现），就传回TRUE，否则传回FALSE：

```
// in header file
class CObject
{
public:
  ...
  BOOL IsKindOf(const CRuntimeClass* pClass) const;
};

// in implementation file
BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
{
        CRuntimeClass* pClassThis = GetRuntimeClass();
        while (pClassThis != NULL)
        {
              if (pClassThis == pClass)
                    return TRUE;
              pClassThis = pClassThis->m_pBaseClass;
        }
        return FALSE;      // walked to the top, no match
}
```

注意，while 循环中所追踪的是「同宗」路线，也就是凭借着m_pBaseClass 而非m_pNextClass。假设我们的调用是：

```
CView* pView = new CView;
pView->IsKindOf(RUNTIME_CLASS(CWinApp));
```

IsKindOf 的参数其实就是&CWinApp::classCWinApp。函数内利用 GetRuntimeClass先取得&CView::classCView，然后循线而上（从图3-1来看，所谓循线分别是指CView、CWnd、CCmdTarget、CObject），每获得一个 CRuntimeClass 对象指针，就拿来和CView::classCView 的指针比对。靠这个土方法，完成了IsKindOf 能力。

2.IsKindOf 的使用方式如下：

```
CMyDoc* pMyDoc = new CMyDoc;
CMyView* pMyView = new CMyView;

cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CMyDoc));          // 应该获得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CDocument));        // 应该获得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CCmdTarget));   // 应该获得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CObject));          // 应该获得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CWinApp));          // 应该获得 FALSE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CView));            // 应该获得 FALSE

cout << pMyView->IsKindOf(RUNTIME_CLASS(CView));           // 应该获得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CObject));         // 应该获得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CWnd));            // 应该获得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CFrameWnd));   // 应该获得 FALSE
```

IsKindOf 的完整范例放在Frame4中。

Frame4 范例程序

Frame4 与 Frame3 大同小异，唯一不同的就是前面所说的，在 CObject 中加上 IsKindOf 函数的声明与定义，并将私有类（non-MFC 类）也挂到「类型录网」中：

```
// in header file
class CMyFrameWnd : public CFrameWnd
{
        DECLARE_DYNAMIC(CMyFrameWnd) // 在 MFC 程序中这里其实是 DECLARE_DYNCREATE()
...                                      // 稍后我便会模拟 DECLARE_DYNCREATE() 给你看
};
class CMyDoc : public CDocument
{
        DECLARE_DYNAMIC(CMyDoc)   // 在 MFC 程序中这里其实是 DECLARE_DYNCREATE()
...                                   // 稍后我便会模拟 DECLARE_DYNCREATE() 给你看
};
class CMyView : public CView
{
        DECLARE_DYNAMIC(CMyView) // 在 MFC 程序中这里其实是 DECLARE_DYNCREATE()
...                                   // 稍后我便会模拟 DECLARE_DYNCREATE() 给你看
};

// in implementation file
...
IMPLEMENT_DYNAMIC(CMyFrameWnd, CFrameWnd) // 在 MFC 程序中这里其实是 IMPLEMENT_DYNCREATE()
                                          // 稍后我便会模拟 IMPLEMENT_DYNCREATE() 给你看
...
IMPLEMENT_DYNAMIC(CMyDoc, CDocument) // 在 MFC 程序中这里其实是 IMPLEMENT_DYNCREATE()
                                     // 稍后我便会模拟 IMPLEMENT_DYNCREATE() 给你看
...
IMPLEMENT_DYNAMIC(CMyView, CView)       // 在 MFC 程序中这里其实是 IMPLEMENT_DYNCREATE()
                                        // 稍后我便会模拟 IMPLEMENT_DYNCREATE() 给你看
```

我不在此列出Frame4的原始代码，你可以在书附光盘片中找到完整的文件。Frame4 的命令列编译链接动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

以下即是 Frame4 的执行结果：

```
pMyDoc->IsKindOf(RUNTIME_CLASS(CMyDoc))          1
pMyDoc->IsKindOf(RUNTIME_CLASS(CDocument))       1
pMyDoc->IsKindOf(RUNTIME_CLASS(CCmdTarget))      1
pMyDoc->IsKindOf(RUNTIME_CLASS(CObject))         1
pMyDoc->IsKindOf(RUNTIME_CLASS(CWinApp))         0
pMyDoc->IsKindOf(RUNTIME_CLASS(CView))           0

pMyView->IsKindOf(RUNTIME_CLASS(CView))          1
pMyView->IsKindOf(RUNTIME_CLASS(CObject))        1
pMyView->IsKindOf(RUNTIME_CLASS(CWnd))           1
pMyView->IsKindOf(RUNTIME_CLASS(CFrameWnd))      0

pMyWnd->IsKindOf(RUNTIME_CLASS(CFrameWnd))       1
pMyWnd->IsKindOf(RUNTIME_CLASS(CWnd))            1
pMyWnd->IsKindOf(RUNTIME_CLASS(CObject))         1
pMyWnd->IsKindOf(RUNTIME_CLASS(CDocument))       0
```

Dynamic Creation（动态生成）

基础有了，做什么都好。同样地，有了上述的「类型录网」，各种应用纷至沓来。其中一个应用就是解决棘手的动态生成问题。

我已经在第二章描述过动态生成的困难点：你没有办法在程序执行期间，根据动态获得的一个类名称（通常来自读文件，但我将以屏幕输入为例），要求程序产生一个对象。

上述的「类型录网」虽然透露出解决此一问题的些微曙光，但是技术上还得加把劲儿。

如果我能够把类的大小记录在类型录中，把构造函数（注意，这里并非指 C++ 构造函数，而是指即将出现的 CRuntimeClass::CreateObject）也记录在类型录中，当程序在执行时期获得一个类名称，它就可以在「类型录网」中找出对应的元素，然后调用其构造函数（这里并非指 C++ 构造函数），产生出对象。

类型录网的元素型式CRuntimeClass 于是有了变化：

```
// in MFC.H
struct CRuntimeClass
{
// Attributes
        LPCSTR m_lpszClassName;
        int m_nObjectSize;
        UINT m_wSchema; // schema number of the loaded class
        CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
        CRuntimeClass* m_pBaseClass;

        CObject* CreateObject();
        static CRuntimeClass* PASCAL Load();

        // CRuntimeClass objects linked together in simple list
        static CRuntimeClass* pFirstClass; // start of class list
        CRuntimeClass* m_pNextClass;      // linked list of registered classes
};
```

DECLARE_DYNCREATE / IMPLEMENT_DYNCREATE 宏

为了因 应CRuntimeClass 中新增的成员变量， 我们再添两个宏， DECLARE_DYNCREATE 和IMPLEMENT_DYNCREATE：

```
#define DECLARE_DYNCREATE(class_name) \
        DECLARE_DYNAMIC(class_name) \
        static CObject* PASCAL CreateObject();

#define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
        CObject* PASCAL class_name::CreateObject() \
                { return new class_name; } \
        _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, \
                class_name::CreateObject)
```

于是，以 CFrameWnd 为例，下列程序代码：

```
// in header file
class CFrameWnd : public CWnd
{
    DECLARE_DYNCREATE(CFrameWnd)
    ...
};
// in implementation file
IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)
```

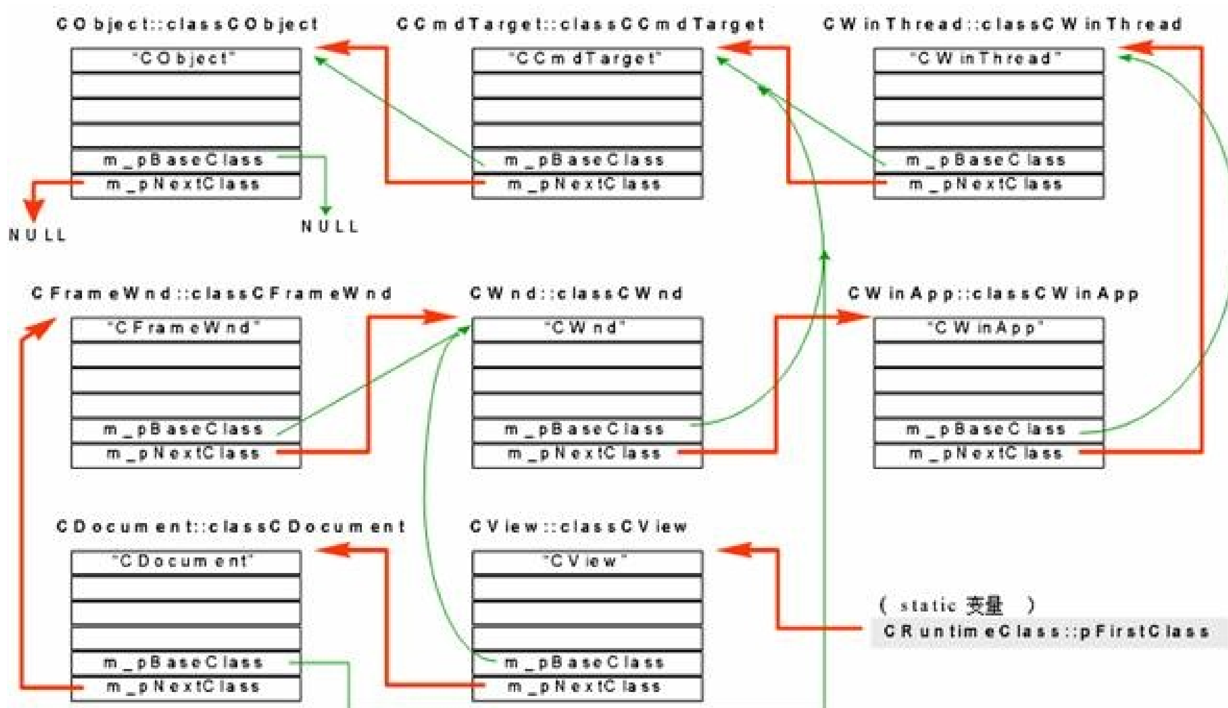
就被展开如下（注意，编译器选项 /P可得前置处理结果）：

```
// in header file
class CFrameWnd : public CWnd
{
public:
        static CRuntimeClass classCFrameWnd;
        virtual CRuntimeClass* GetRuntimeClass() const;
        static CObject* PASCAL CreateObject();
    ...
};

// in implementation file
CObject* PASCAL CFrameWnd::CreateObject()
        { return new CFrameWnd; }

static char _lpszCFrameWnd[] = "CFrameWnd";
CRuntimeClass CFrameWnd::classCFrameWnd = {
        _lpszCFrameWnd, sizeof(CFrameWnd), 0xFFFF, CFrameWnd::CreateObject,
                RUNTIME_CLASS(CWnd), NULL };
static AFX_CLASSINIT _init_CFrameWnd(&CFrameWnd::classCFrameWnd);
CRuntimeClass* CFrameWnd::GetRuntimeClass() const
        { return &CFrameWnd::classCFrameWnd; }
```

图示如下：



「对象生成器」CreateObject 函数很简单，只要说new 就好。

从宏的定义我们很清楚可以看出，拥有动态生成（Dynamic Creation）能力的类库，必然亦拥有运行时类型识别（RTTI）能力，因为_DYNCREATE宏涵盖了_DYNAMIC宏。

范例程序 Frame6 在 .h 文件中有这些类声明：

```
class CObject
{
...
};
class CCmdTarget : public CObject
{
        DECLARE_DYNAMIC(CCmdTarget)
...
};
class CWinThread : public CCmdTarget
{
        DECLARE_DYNAMIC(CWinThread)
...
};
class CWinApp : public CWinThread
{
        DECLARE_DYNAMIC(CWinApp)
...
};
class CDocument : public CCmdTarget
{
        DECLARE_DYNAMIC(CDocument)
...
};
```

```
class CWnd : public CCmdTarget
{
        DECLARE_DYNCREATE(CWnd)
...
};
class CFrameWnd : public CWnd
{
        DECLARE_DYNCREATE(CFrameWnd)
...
};
class CView : public CWnd
{
        DECLARE_DYNAMIC(CView)
...
};
class CMyWinApp : public CWinApp
{
...
};
class CMyFrameWnd : public CFrameWnd
```

```
{
        DECLARE_DYNCREATE(CMyFrameWnd)
...
};
class CMyDoc : public CDocument
{
        DECLARE_DYNCREATE(CMyDoc)
...
};
class CMyView : public CView
{
        DECLARE_DYNCREATE(CMyView)
...
};
```

在 .cpp文件中又有这些动作：

```
IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
IMPLEMENT_DYNCREATE(CWnd, CCmdTarget)
IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)
IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
IMPLEMENT_DYNAMIC(CView, CWnd)
IMPLEMENT_DYNCREATE(CMyFrameWnd, CFrameWnd)
IMPLEMENT_DYNCREATE(CMyDoc, CDocument)
IMPLEMENT_DYNCREATE(CMyView, CView)
```

于是组织出图 3-2 这样一个大网。

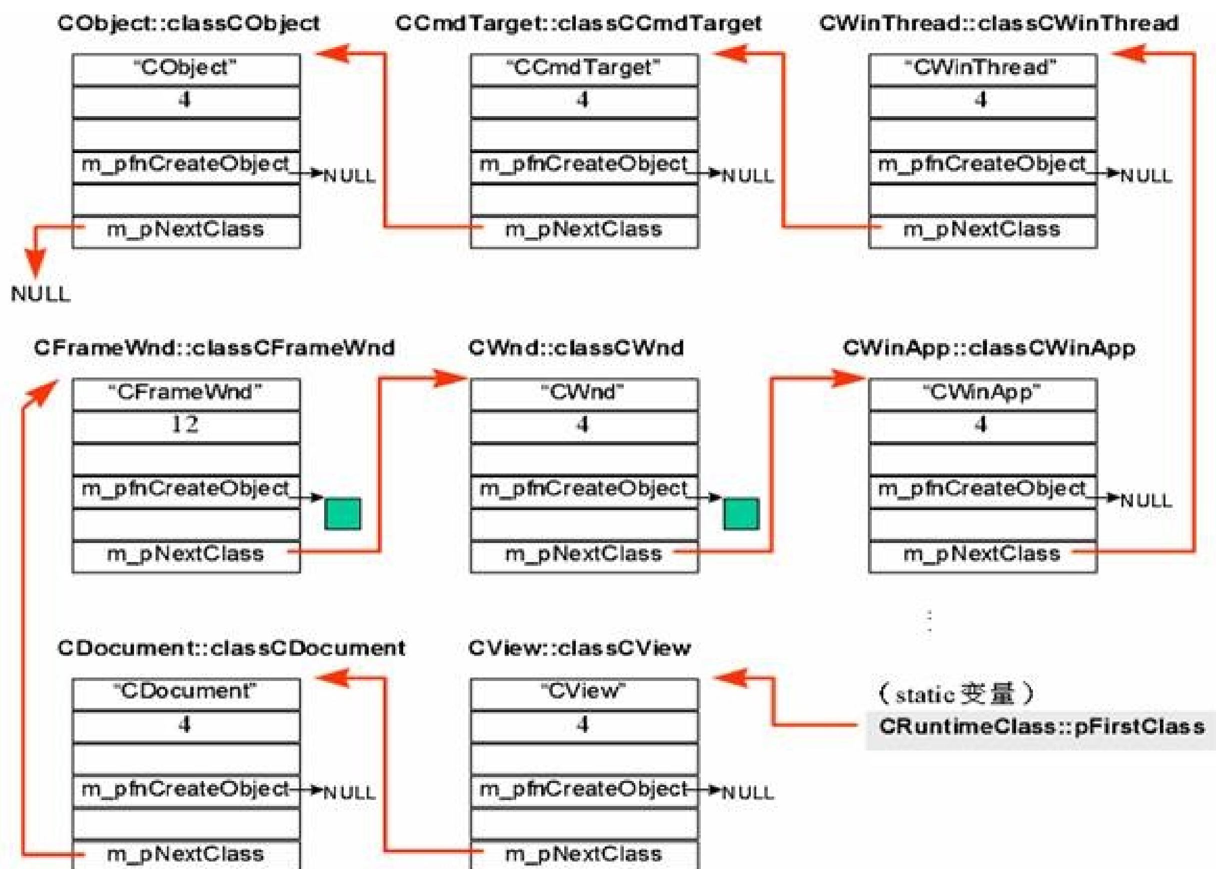


图3-2 以 CRuntimeClass 对象构成的「类型录网」。

本图只列出与动态生成（Dynamic Creation）有关系的成员。

凡是m_pfnCreateObject不为NULL 者，即可动态生成。

现在，我们开始仿真动态生成。首先在main函数中加上这一段代码：

```
void main()
{
  ...
  //Test Dynamic Creation
  CRuntimeClass* pClassRef;
  CObject* pOb;
  while(1)
  {

    if ((pClassRef = CRuntimeClass::Load()) == NULL)
        break;

    pOb = pClassRef->CreateObject();
    if (pOb != NULL)
        pOb->SayHello();
  }
}
```

并设计CRuntimeClass::CreateObject和CRuntimeClass::Load如下：

```
// in implementation file
CObject* CRuntimeClass::CreateObject()
{
        if (m_pfnCreateObject == NULL)
        {
                TRACE1("Error: Trying to create object which is not "
                        "DECLARE_DYNCREATE \nor DECLARE_SERIAL: %hs.\n",
                         m_lpszClassName);
                return NULL;
        }
```

```
        CObject* pObject = NULL;
        pObject = (*m_pfnCreateObject)();

        return pObject;
}

CRuntimeClass* PASCAL CRuntimeClass::Load()
{
        char szClassName[64];
        CRuntimeClass* pClass;
        // JJHOU : instead of Load from file, we Load from cin.
        cout << "enter a class name...  ";
        cin >> szClassName;

        for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
        {
                if (strcmp(szClassName, pClass->m_lpszClassName) == 0)
                        return pClass;
        }

        TRACE1("Error: Class not found: %s \n", szClassName);
        return NULL; // not found
}
```

然后，为了验证这样的动态生成机制的确有效（也就是对象的确被产生了），我让许多个类的构造函数都输出一段文字，而且在取得对象指针后，真的去调用该对象的一个成员函数SayHello。我把SayHello设计为虚函数，所以根据不同的对象类型，会调用到不同的SayHello函数，出现不同的输出字符串。

请注意，main 函数中的while循环必须等到CRuntimeClass::Load传回 NULL才会停止，而CRuntimeClass::Load 是在它从整个「类型录网」中找不到它要找的那个类名称时，才传回NULL。这些都是我为了模拟与示范，所采取的权宜设计。

Frame6 的命令列编译链接动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

下面是 Frame6 的执行结果。粗体表示我（程序执行者）在屏幕上输入的类名称：

```
enter a class name...  CObject
Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CObject.

enter a class name...  CView
Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CView.

enter a class name...  CMyView
CWnd Constructor
CMyView Constructor
Hello CMyView

enter a class name...  CMyFrameWnd
CWnd Constructor
CFrameWnd Constructor
CMyFrameWnd Constructor
Hello CMyFrameWnd

enter a class name...  CMyDoc
CMyDoc Constructor
Hello CMyDoc
enter a class name...  CWinApp
Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CWinApp.

enter a class name...  CJjhou  （故意输入一个不在「类型录网」中的类名称）
Error: Class not found: CJjhou  （程序结束）
```

Frame6 范例程序

MFC.H

```
#0001  #define BOOL int
#0002  #define TRUE 1
#0003  #define FALSE 0
#0004  #define LPCSTR  LPSTR
#0005  typedef char*   LPSTR;
#0006  #define UINT int
#0007  #define PASCAL _stdcall
#0008  #define TRACE1 printf
#0009
#0010  #include <iostream.h>
#0011  #include <stdio.h>
#0012  #include <string.h>
#0013
#0014  class CObject;
#0015
#0016  struct CRuntimeClass
#0017  {
#0018  // Attributes
#0019     LPCSTR m_lpszClassName;
#0020     int m_nObjectSize;
#0021     UINT m_wSchema; // schema number of the loaded class
#0022     CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
#0023     CRuntimeClass* m_pBaseClass;
#0024
#0025     CObject* CreateObject();
#0026     static CRuntimeClass* PASCAL Load();
#0027
#0028     // CRuntimeClass objects linked together in simple list
#0029     static CRuntimeClass* pFirstClass; // start of class list
#0030     CRuntimeClass* m_pNextClass;    // linked list of registered classes
#0031  };
#0032
#0033  struct AFX_CLASSINIT
#0034          { AFX_CLASSINIT(CRuntimeClass* pNewClass); };
#0035
#0036  #define RUNTIME_CLASS(class_name) \
#0037          (&class_name::class##class_name)
#0038
#0039  #define DECLARE_DYNAMIC(class_name) \
#0040  public: \
#0041          static CRuntimeClass class##class_name; \
#0042          virtual CRuntimeClass* GetRuntimeClass() const;
#0043
#0044  #define DECLARE_DYNCREATE(class_name) \
#0045          DECLARE_DYNAMIC(class_name) \
#0046          static CObject* PASCAL CreateObject();
```

```
#0047
#0048 #define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
#0049         static char _lpsz##class_name[] = #class_name; \
#0050         CRuntimeClass class_name::class##class_name = { \
#0051                 _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
#0052                         RUNTIME_CLASS(base_class_name), NULL }; \
#0053         static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
#0054         CRuntimeClass* class_name::GetRuntimeClass() const \
#0055                 { return &class_name::class##class_name; } \
#0056
#0057 #define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
#0058         _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#0059
#0060 #define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
#0061         CObject* PASCAL class_name::CreateObject() \
#0062                 { return new class_name; } \
#0063         _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, \
#0064                 class_name::CreateObject)
#0065
#0066 class CObject
#0067 {
#0068 public:
#0069   CObject::CObject()  {
#0070                       }
#0071   CObject::~CObject() {
#0072                        }
#0073
#0074   virtual CRuntimeClass* GetRuntimeClass() const;
#0075   BOOL IsKindOf(const CRuntimeClass* pClass) const;
#0076
#0077 public:
#0078   static CRuntimeClass classCObject;
#0079   virtual void SayHello() { cout << "Hello CObject \n"; }
#0080 };
#0081
#0082 class CCmdTarget : public CObject
#0083 {
#0084         DECLARE_DYNAMIC(CCmdTarget)
#0085 public:
#0086   CCmdTarget::CCmdTarget()  {
#0087                             }
#0088   CCmdTarget::~CCmdTarget() {
#0089                             }
#0090 };
#0091
#0092 class CWinThread : public CCmdTarget
#0093 {
#0094         DECLARE_DYNAMIC(CWinThread)
#0095 public:
#0096   CWinThread::CWinThread()  {
#0097                             }
#0098   CWinThread::~CWinThread() {
#0099                             }
#0100
```

```
#0101   virtual BOOL InitInstance() {
#0102                                   return TRUE;
#0103                                 }
#0104   virtual int Run() {
#0105                         return 1;
#0106                       }
#0107 };
#0108
#0109 class CWnd;
#0110
#0111 class CWinApp : public CWinThread
#0112 {
#0113         DECLARE_DYNAMIC(CWinApp)
#0114 public:
#0115   CWinApp* m_pCurrentWinApp;
#0116   CWnd* m_pMainWnd;
#0117
#0118 public:
#0119   CWinApp::CWinApp()  {
#0120                         m_pCurrentWinApp = this;
#0121                       }
#0122   CWinApp::~CWinApp() {
#0123                        }
#0124
#0125   virtual BOOL InitApplication() {
#0126                                      return TRUE;
#0127                                    }
#0128   virtual BOOL InitInstance()    {
#0129                                      return TRUE;
#0130                                    }
#0131   virtual int Run() {
#0132                         return CWinThread::Run();
#0133                       }
#0134 };
#0135
#0136
#0137 class CDocument : public CCmdTarget
#0138 {
#0139         DECLARE_DYNAMIC(CDocument)
#0140 public:
#0141   CDocument::CDocument()    {
#0142                             }
#0143   CDocument::~CDocument()   {
#0144                             }
#0145 };
#0146
#0147 class CWnd : public CCmdTarget
#0148 {
#0149         DECLARE_DYNCREATE(CWnd)
#0150 public:
#0151   CWnd::CWnd()   {
#0152                   cout << "CWnd Constructor \n";
#0153                  }
#0154   CWnd::~CWnd()  {
#0155                  }
```

```
#0156
#0157   virtual BOOL Create();
#0158   BOOL CreateEx();
#0159   virtual BOOL PreCreateWindow();
#0160   void SayHello() { cout << "Hello CWnd \n"; }
#0161 };
#0162
#0163 class CFrameWnd : public CWnd
#0164 {
#0165         DECLARE_DYNCREATE(CFrameWnd)
#0166 public:
#0167   CFrameWnd::CFrameWnd()   {
#0168                             cout << "CFrameWnd Constructor \n";
#0169                            }
#0170   CFrameWnd::~CFrameWnd()  {
#0171                            }
#0172   BOOL Create();
#0173   virtual BOOL PreCreateWindow();
#0174   void SayHello() { cout << "Hello CFrameWnd \n"; }
#0175 };
#0176
#0177 class CView : public CWnd
#0178 {
#0179         DECLARE_DYNAMIC(CView)
#0180 public:
#0181   CView::CView()   {
#0182                    }
#0183   CView::~CView()  {
#0184                    }
#0185 };
#0186
#0187 // global function
#0188 CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001 #include "my.h"  // it should be mfc.h, but for CMyWinApp definition, so...
#0002
#0003 extern CMyWinApp theApp;
#0004
#0005 static char szCObject[] = "CObject";
#0006 struct CRuntimeClass CObject::classCObject =
#0007         { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
#0008 static AFX_CLASSINIT _init_CObject(&CObject::classCObject);
#0009
#0010 CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
#0011
#0012 AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
#0013 {
#0014         pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
#0015         CRuntimeClass::pFirstClass = pNewClass;
#0016 }
#0017
#0018 CObject* CRuntimeClass::CreateObject()
#0019 {
#0020         if (m_pfnCreateObject == NULL)
#0021         {
#0022                 TRACE1("Error: Trying to create object which is not "
#0023                         "DECLARE_DYNCREATE \nor DECLARE_SERIAL: %hs.\n",
#0024                         m_lpszClassName);
#0025                 return NULL;
#0026         }
```

```
#0027
#0028         CObject* pObject = NULL;
#0029         pObject = (*m_pfnCreateObject)();
#0030
#0031         return pObject;
#0032 }
#0033
#0034 CRuntimeClass* PASCAL CRuntimeClass::Load()
#0035 {
#0036         char szClassName[64];
#0037         CRuntimeClass* pClass;
#0038
#0039         // JJHOU : instead of Load from file, we Load from cin.
#0040         cout << "enter a class name...  ";
#0041         cin >> szClassName;
#0042
#0043         for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
#0044         {
#0045                 if (strcmp(szClassName, pClass->m_lpszClassName) == 0)
#0046                         return pClass;
#0047         }
#0048
#0049         TRACE1("Error: Class not found: %s \n", szClassName);
#0050         return NULL; // not found
#0051 }
#0052
#0053 CRuntimeClass* CObject::GetRuntimeClass() const
#0054 {
#0055         return &CObject::classCObject;
#0056 }
#0057
#0058 BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
#0059 {
#0060         CRuntimeClass* pClassThis = GetRuntimeClass();
#0061         while (pClassThis != NULL)
#0062         {
#0063                 if (pClassThis == pClass)
#0064                         return TRUE;
#0065                 pClassThis = pClassThis->m_pBaseClass;
#0066         }
#0067         return FALSE;       // walked to the top, no match
#0068 }
#0069
#0070 BOOL CWnd::Create()
#0071 {
#0072   return TRUE;
#0073 }
#0074
#0075 BOOL CWnd::CreateEx()
```

```
#0076  {
#0077    PreCreateWindow();
#0078    return TRUE;
#0079  }
#0080
#0081  BOOL CWnd::PreCreateWindow()
#0082  {
#0083    return TRUE;
#0084  }
#0085
#0086  BOOL CFrameWnd::Create()
#0087  {
#0088    CreateEx();
#0089    return TRUE;
#0090  }
#0091
#0092  BOOL CFrameWnd::PreCreateWindow()
#0093  {
#0094    return TRUE;
#0095  }
#0096
#0097  CWinApp* AfxGetApp()
#0098  {
#0099    return theApp.m_pCurrentWinApp;
#0100  }
#0101
#0102  IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
#0103  IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
#0104  IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
#0105  IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
#0106  IMPLEMENT_DYNCREATE(CWnd, CCmdTarget)
#0107  IMPLEMENT_DYNAMIC(CView, CWnd)
#0108  IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()   {
#0008                              }
#0009    CMyWinApp::~CMyWinApp()  {
#0010                              }
#0011
#0012    virtual BOOL InitInstance();
#0013  };
#0014
#0015  class CMyFrameWnd : public CFrameWnd
#0016  {
#0017          DECLARE_DYNCREATE(CMyFrameWnd)
#0018  public:
```

```
#0019   CMyFrameWnd();
#0020   ~CMyFrameWnd()  {
#0021                   }
#0022   void SayHello() { cout << "Hello CMyFrameWnd \n"; }
#0023  };
#0024
#0025  class CMyDoc : public CDocument
#0026  {
#0027          DECLARE_DYNCREATE(CMyDoc)
#0028  public:
#0029   CMyDoc::CMyDoc()  {
#0030                       cout << "CMyDoc Constructor \n";
#0031                     }
#0032   CMyDoc::~CMyDoc() {
#0033                     }
#0034   void SayHello() { cout << "Hello CMyDoc \n"; }
#0035  };
#0036
#0037  class CMyView : public CView
#0038  {
#0039          DECLARE_DYNCREATE(CMyView)
#0040  public:
#0041   CMyView::CMyView()   {
#0042                          cout << "CMyView Constructor \n";
#0043                        }
#0044   CMyView::~CMyView()  {
#0045                         }
#0046   void SayHello() { cout << "Hello CMyView \n"; }
#0047  };
#0048
#0049  // global function
#0050  void AfxPrintAllClasses();
```

MY.CPP

```
#0001  #include "my.h"
#0002
#0003  CMyWinApp theApp;
#0004
#0005  BOOL CMyWinApp::InitInstance()
#0006  {
#0007      m_pMainWnd = new CMyFrameWnd;
#0008      return TRUE;
#0009  }
#0010
#0011  CMyFrameWnd::CMyFrameWnd()
#0012  {
#0013      cout << "CMyFrameWnd Constructor \n";
#0014      Create();
#0015  }
#0016
#0017  IMPLEMENT_DYNCREATE(CMyFrameWnd, CFrameWnd)
#0018  IMPLEMENT_DYNCREATE(CMyDoc, CDocument)
#0019  IMPLEMENT_DYNCREATE(CMyView, CView)
#0020
```

```
#0021 void PrintAllClasses()
#0022 {
#0023     CRuntimeClass* pClass;
#0024
#0025     // just walk through the simple list of registered classes
#0026     for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
#0027             pClass = pClass->m_pNextClass)
#0028     {
#0029         cout << pClass->m_lpszClassName << "\n";
#0030         cout << pClass->m_nObjectSize << "\n";
#0031         cout << pClass->m_wSchema << "\n";
#0032     }
#0033 }
#0034 //------------------------------------------------------------------
#0035 // main
#0036 //------------------------------------------------------------------
#0037 void main()
#0038 {
#0039   CWinApp* pApp = AfxGetApp();
#0040
#0041   pApp->InitApplication();
#0042   pApp->InitInstance();
#0043   pApp->Run();
#0044
#0045   //Test Dynamic Creation
#0046   CRuntimeClass* pClassRef;
#0047   CObject* pOb;
#0048   while(1)
#0049   {
#0050     if ((pClassRef = CRuntimeClass::Load()) == NULL)
#0051         break;
#0052
#0053     pOb = pClassRef->CreateObject();
#0054     if (pOb != NULL)
#0055         pOb->SayHello();
#0056   }
#0057 }
```

Persistence（永续生存）机制

面向对象有一个术语：Persistence，意思就是把对象永久保留下来。Power 一关，啥都没有，对象又如何能够永续存留？当然是写到文件去啰。

把数据写到文件，很简单。在Document/View架构中，数据都放在一份 document（文件）里头，我们只要把其中的成员变量依续写进文件即可。成员变量很可能是个对象，而面对对象，我们首先应该记载其类名称，然后才是对象中的数据。

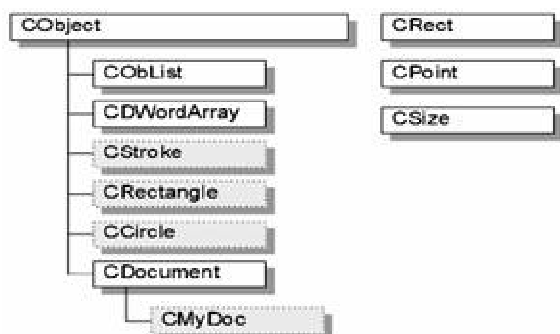
读文件就有点麻烦了。当程序从文件中读到一个类名称，它如何实现（instantiate）一个对象？呵，这不就是动态生成的技术吗？我们在前一章已经解决掉了。

MFC 有一套Serialize机制，目的在于把文件名的选择、文件的开关、缓冲区的建立、数据的读写、提取运算符（ `>>` ）和嵌入运算符（ `<<` ）的重载（overload）、对象的动态生成...都包装起来。

上述Serialize 的各部分工作，除了数据的读写和对象的动态生成，其余都是支节。动态生成的技术已经解决，让我们集中火力，分析数据的读写动作。

Serialize （数据读写）

假设我有一份文件，用以记录一张图形。图形只有三种基本元素：线条（Stroke）、圆形、矩形。我打算用以下类，组织这份文件：



其中CObList和CDWordArray是MFC提供的类，前者是一个串列，可放置任何从CObject 派生下来的对象，后者是一个数组，每一个元素都是 "double word"。另外三个类：CStroke 和 CRectangle 和 CCircle，是我从 CObject 中派生下来的类。

```
class CMyDoc : public CDocument
{
    CObList m_graphList;
    CSize   m_sizeDoc;
    ...
};
class CStroke : public CObject
{
    CDWordArray m_ptArray;   // series of connected points
    ...
};
class CRectangle : public CObject
{
    CRect m_rect;
    ...
};
class CCircle : public CObject

{
   CPoint  m_center;
   UINT    m_radius;
   ...
};
```

假设现有一份文件，内容如图 3-3，如果你是Serialize 机制的设计者，你希望怎么做呢？

把图 3-3 写成这样的文件内容好吗：

```
06 00                          ;CObList elements count

07 00                          ;class name string length
43 53 74 72 6F 6B 65           ;"CStroke"
02 00                          ;DWordArray size
28 00 13 00                    ;point
28 00 13 00                    ;point

0A 00                          ;class name string length
43 52 65 63 74 61 6E 67 6C 65  ;"CRectangle"
11 00 22 00 33 00 44 00        ;CRect

07 00                          ;class name string length
43 43 69 72 63 6C 65           ;"CCircle"
55 00 66 00 77 00              ;CPoint & radius

07 00                          ;class name string length
43 53 74 72 6F 6B 65           ;"CStroke"
02 00                          ;DWordArray size
28 00 35 00                    ;point
28 00 35 00                    ;point

0A 00                          ;class name string length
43 52 65 63 74 61 6E 67 6C 65  ;"CRectangle"
11 00 22 00 33 00 44 00        ;CRect

07 00                           ;class name string length
43 43 69 72 63 6C 65            ;"CCircle"
55 00 66 00 77 00               ;CPoint & radius
```

还算堪用。但如果考虑到屏幕卷动的问题，以及印表输出的问题，应该在最前端增加「文件大小」。另外，如果这份文件有 100 条线条，50 个圆形，80 个矩形，难不成我们要记录 230 个类名称？应该有更好的方法才是。

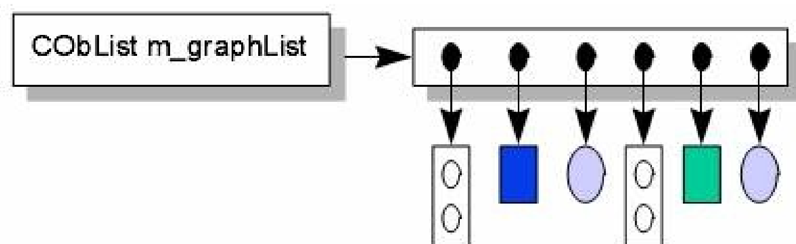


图3-3 一个串列，内含三种基本图形：线条、圆形、矩形。

我们可以在每次记录对象内容的时候，先写入一个代码，表示此对象之类是否曾在文件中记录过了。如果是新类，乖乖地记录其类名称；如果是旧类，则以代码表示。

这样可以节省文件大小以及程序用于解析的时间。啊，不要看到文件大小就想到硬盘很便宜，桌上的一切都将被带到网上，你得想想网络频宽这回事。

还有一个问题。文件的「版本」如何控制？旧版程序读取新版文件，新版程序读取旧版文件，都可能出状况。为了防弊，最好把版本号代码记录上去。最好是每个类有自己的版本号代码。

下面是新的构想，也就是Serialization 的目标：

```
20 03 84 03                     ;Document Size
06 00                           ;CObList elements count

FF FF                           ;new class tag
02 00                           ;schema
07 00                           ;class name string length
43 53 74 72 6F 6B 65            ;"CStroke"
02 00                           ;DWordArray size
28 00 13 00                     ;point
28 00 13 00                     ;point

FF FF                               ;new class tag
01 00                               ;schema
0A 00                               ;class name string length
43 52 65 63 74 61 6E 67 6C 65       ;"CRectangle"
11 00 22 00 33 00 44 00             ;CRect

FF FF                               ;new class tag
01 00                               ;schema
07 00                               ;class name string length
43 43 69 72 63 6C 65                ;"CCircle"
55 00 66 00 77 00                   ;CPoint & radius

01 80                               ;old class tag
02 00                               ;DWordArray size
28 00 35 00                         ;point
28 00 35 00                         ;point

03 80                               ;old class tag
11 00 22 00 33 00 44 00             ;CRect

05 80                               ;old class tag
55 00 66 00 77 00                   ;CPoint & radius
```

我希望有一个专门负责 Serialization的函数，就叫作 Serialize好了。假设现在我的 Document 类名称为 CScribDoc，我希望有这么便利的程序方法（请仔细琢磨琢磨其便利性）：

```
void CScribDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_sizeDoc;
    else
        ar >> m_sizeDoc;
    m_graphList.Serialize(ar);
}

void CObList::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {
```

```
        ar << (WORD) m_nCount;
        for (CNode* pNode = m_pNodeHead; pNode != NULL; pNode = pNode->pNext)
            ar << pNode->data;
    }
    else {
        WORD nNewCount;
        ar >> nNewCount;
        while (nNewCount--) {
            CObject* newData;
            ar >> newData;
            AddTail(newData);
        }
    }
}

void CStroke::Serialize(CArchive& ar)
{
    m_ptArray.Serialize(ar);
}

void CDWordArray::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {
        ar << (WORD) m_nSize;
        for (int i = 0; i < m_nSize; i++)
            ar << m_pData[i];
    }
    else {
        WORD nOldSize;
        ar >> nOldSize;
        for (int i = 0; i < m_nSize; i++)
            ar >> m_pData[i];
    }
}

void CRectangle::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_rect;
    else
        ar >> m_rect;
}

void CCircle::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {

        ar << (WORD)m_center.x;
        ar << (WORD)m_center.y;
        ar << (WORD)m_radius;
    }
    else {
        ar >> (WORD&)m_center.x;
        ar >> (WORD&)m_center.y;
        ar >> (WORD&)m_radius;
    }
}
```

每一个可写到文件或可从文件中读出的类，都应该有它自己的 Serailize 函数，负责它自己的数据读写文件动作。此类并且应该改写 `<<` 运算符和 `>>` 运算符，把数据导流到archive中。archive 是什么？是一个与文件息息相关的缓冲区，暂时你可以想象它就是文件的化身。当图 3-3 的文件写入文件时，Serialize 函数的调用次序如图 3-4。

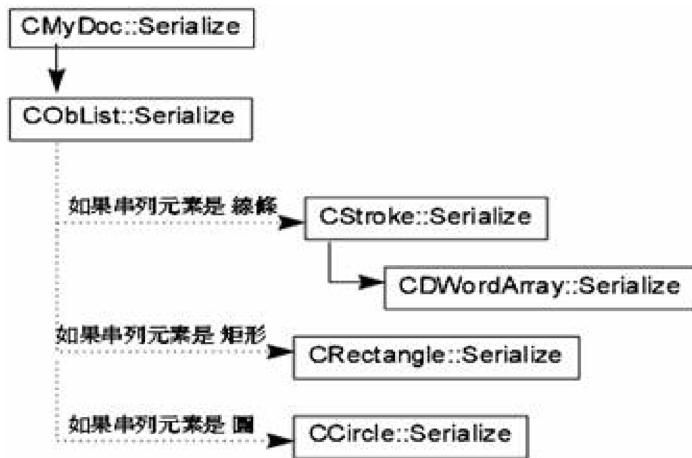


图3-4 图3-3的文件内容写入文件时，Serialize 函数的调用次序。

DECLARE_SERIAL / IMPLEMENT_SERIAL 宏

要将 `<<` 和 `>>` 两个运算符重载化，还要让 Serialize 函数神不知鬼不觉地放入类声明之中，最好的作法仍然是使用宏。

类之能够进行文件读写动作，前提是拥有动态生成的能力，所以，MFC 设计了两个宏 DECLARE_SERIAL和IMPLEMENT_SERIAL：

```
#define DECLARE_SERIAL(class_name) \
        DECLARE_DYNCREATE(class_name) \
        friend CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb);

#define IMPLEMENT_SERIAL(class_name, base_class_name, wSchema) \
        CObject* PASCAL class_name::CreateObject() \
                { return new class_name; } \
        _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, \
                class_name::CreateObject) \
        CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb) \
              { pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
                        return ar; } \
```

为了在每一个对象被处理（读或写）之前，能够处理琐屑的工作，诸如判断是否第一次出现、记录版本号代码、记录文件名等工作，CRuntimeClass 需要两个函数Load 和Store：

```
struct CRuntimeClass
{
// Attributes
        LPCSTR m_lpszClassName;
        int m_nObjectSize;
        UINT m_wSchema; // schema number of the loaded class
        CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
        CRuntimeClass* m_pBaseClass;

        CObject* CreateObject();
        void Store(CArchive& ar) const;
        static CRuntimeClass* PASCAL Load(CArchive& ar, UINT* pwSchemaNum);

        // CRuntimeClass objects linked together in simple list
        static CRuntimeClass* pFirstClass; // start of class list
        CRuntimeClass* m_pNextClass;     // linked list of registered classes
};
```

你已经在上一节看过Load 函数，当时为了简化，我把它的参数拿掉，改为由屏幕上获得类名称，事实上它应该是从文件中读一个类名称。至于Store函数，是把类名称写入文件中：

```
// Runtime class serialization code
```

```
// Runtime class serialization code
CRuntimeClass* PASCAL CRuntimeClass::Load(CArchive& ar, UINT* pwSchemaNum)
{
    WORD nLen;
    char szClassName[64];
    CRuntimeClass* pClass;

    ar >> (WORD&)(*pwSchemaNum) >> nLen;

    if (nLen >= sizeof(szClassName) || ar.Read(szClassName, nLen) != nLen)
            return NULL;
    szClassName[nLen] = '\0';

    for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
    {
        if (lstrcmp(szClassName, pClass->m_lpszClassName) == 0)
            return pClass;
    }
    return NULL; // not found
}

void CRuntimeClass::Store(CArchive& ar) const
        // stores a runtime class description
{
        WORD nLen = (WORD)lstrlenA(m_lpszClassName);
        ar << (WORD)m_wSchema << nLen;
        ar.Write(m_lpszClassName, nLen*sizeof(char));
}
```

图 3-4 的例子中，为了让整个Serialization机制运作起来，我们必须做这样的类声明：

```
class CScribDoc : public CDocument
{
    DECLARE_DYNCREATE(CScribDoc)
    ...
};
class CStroke : public CObject
{
    DECLARE_SERIAL(CStroke)
public:
    void Serialize(CArchive&);
    ...
};
class CRectangle : public CObject
{
    DECLARE_SERIAL(CRectangle)
public:
    void Serialize(CArchive&);
    ...
};
class CCircle : public CObject
{
    DECLARE_SERIAL(CCircle)
public:
    void Serialize(CArchive&);
    ...
};
```

以及在 .CPP 文件中做这样的动作：

```
IMPLEMENT_DYNCREATE(CScribDoc, CDocument)
IMPLEMENT_SERIAL(CStroke, CObject, 2)
IMPLEMENT_SERIAL(CRectangle, CObject, 1)
IMPLEMENT_SERIAL(CCircle, CObject, 1)
```

然后呢？分头设计CStroke、CRectangle和CCircle的Serialize 函数吧。当然，毫不令人意外地，MFC原始代码中的CObList和CDWordArray有这样的内容：

```
// in header files
class CDWordArray : public CObject
{
    DECLARE_SERIAL(CDWordArray)
public:
    void Serialize(CArchive&);
    ...
};
class CObList : public CObject
{
    DECLARE_SERIAL(CObList)
public:
    void Serialize(CArchive&);
    ...
};
// in implementation files
IMPLEMENT_SERIAL(CObList, CObject, 0)
IMPLEMENT_SERIAL(CDWordArray, CObject, 0)
```

而 CObject 也多了一个虚函数 Serialize：

```
class CObject
{
public:
    virtual void Serialize(CArchive& ar);
    ...
}
```

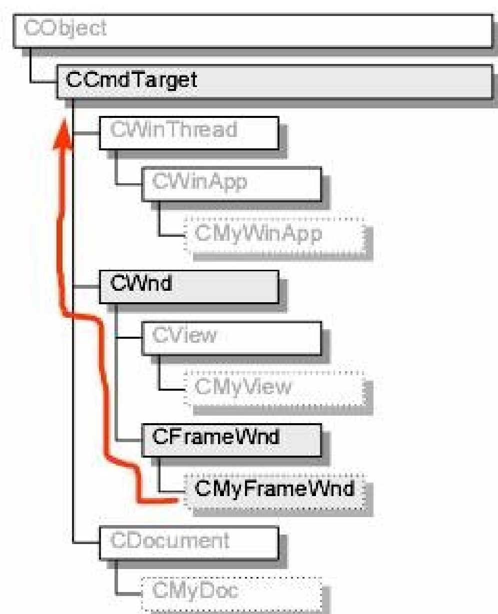
Message Mapping（消息映射）

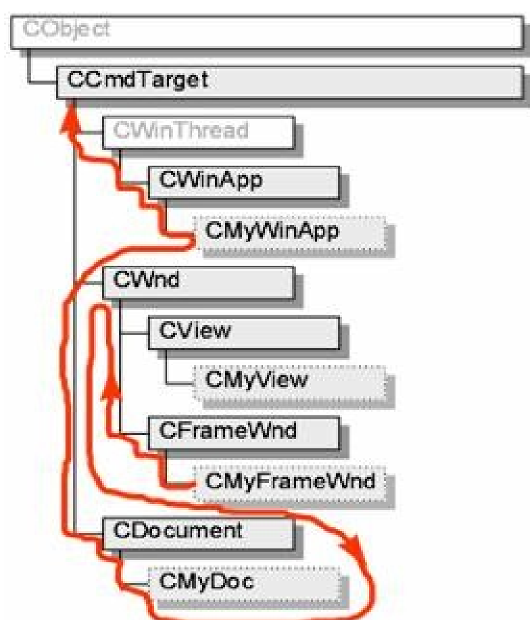
Windows 程序靠消息的流动而维护生命。你已经在第一章看过了消息的一般处理方式，也就是在窗口函数中借着一个大大的 switch/case 比对动作，判别消息再调用对应的处理例程。为了让大大的 switch/case 比对动作简化，也让程序代码更模块化一些，我在第 1 章提供了一个简易的消息映射表作法，把消息和其处理例程关联起来。

当我们的类库成立，如果其中与消息有关的类（姑且叫作「消息标的类」好了，在 MFC 之中就是 CCmdTarget）都是一条鞭式地继承，我们应该为每一个「消息标的类」准备一个消息映射表，并且将基类与派生类之消息映射表串接起来。然后，当窗口函数做消息的比对时，我们就可以想办法导引它沿着这条路走过去：

但是，MFC之中用来处理消息的C++ 类，并不呈单鞭发展。作为application framework的重要架构之一的document/view，也具有处理消息的能力（你现在可能还不清楚什么是document/view，没有关系）。因此，消息藉以攀爬的路线应该有横流的机会：

消息如何流动，我们暂时先不管。是直线前进，或是中途换跑道，我们都暂时不管，本节先把这个攀爬路线网建立起来再说。这整个攀爬路线网就是所谓的消息映射表（Message Map）；说它是一张地图，当然也没有错。将消息与表格中的元素比对，然后调用对应的处理例程，这种动作我们也称之为消息映射（Message Mapping）。





为了尽量降低对正常（一般）类声明和定义的影响，我们希望，最好能够像 RTTI和Dynamic Creation 一样，用一两个宏就完成这巨大蜘蛛网的构造。最好能够像DECLARE_DYNAMIC 和 IMPLEMENT_DYNAMIC 宏那么方便。

首先定义一个数据结构：

```
struct AFX_MSGMAP
{
    AFX_MSGMAP* pBaseMessageMap;
    AFX_MSGMAP_ENTRY* lpEntries;
};
```

其中的AFX_MSGMAP_ENTRY 又是另一个数据结构：

```
struct AFX_MSGMAP_ENTRY  // MFC 4.0 format
{
    UINT nMessage; // windows message
    UINT nCode; // control code or WM_NOTIFY code
    UINT nID; // control ID (or 0 for windows messages)
    UINT nLastID; // used for entries specifying a range of control id's
    UINT nSig; // signature type (action) or pointer to message #
    AFX_PMSG pfn; // routine to call (or special value)
};
```

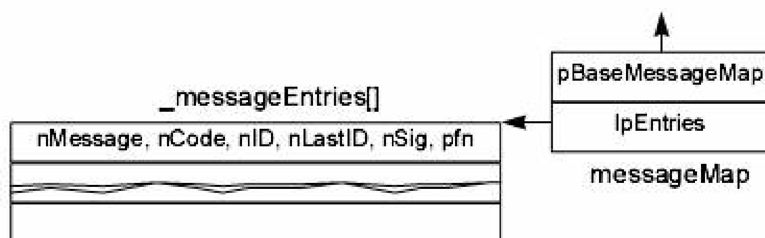
其中的 AFX_PMSG 定义为函数指针：

```
typedef void (CCmdTarget::*AFX_PMSG)(void);
```

然后我们定义一个宏：

```
#define DECLARE_MESSAGE_MAP() \
static AFX_MSGMAP_ENTRY _messageEntries[]; \
static AFX_MSGMAP messageMap; \
virtual AFX_MSGMAP* GetMessageMap() const;
```

于是，DECLARE_MESSAGE_MAP 就相当于声明了这样一个数据结构：



这个数据结构的内容填塞工作由三个宏完成：

```
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
AFX_MSGMAP* theClass::GetMessageMap() const \
{ return &theClass::messageMap; } \
AFX_MSGMAP theClass::messageMap = \
{ &(baseClass::messageMap), \
(AFX_MSGMAP_ENTRY*) &(theClass::_messageEntries) }; \
AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
{
#define ON_COMMAND(id, memberFxn) \
{WM_COMMAND,0,(WORD)id,(WORD)id,AfxSig_vv,(AFX_PMSG)memberFxn},
#define END_MESSAGE_MAP() \
{ 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
};
```

其中的 AfxSig_end 定义为：

```
enum AfxSig
{
    AfxSig_end = 0, // [marks end of message map]
    AfxSig_vv,
};
```

AfxSig_xx 用来描述消息处理例程 memberFxn 的类型（参数与回返值）。本例纯为模拟与简化，所以不在这上面作文章。真正讲到MFC时（第四篇p580），我会再解释它。

于是，以CView为例，下面的原始代码：

```
// in header file
class CView : public CWnd
{
public:
  ...
  DECLARE_MESSAGE_MAP()
};

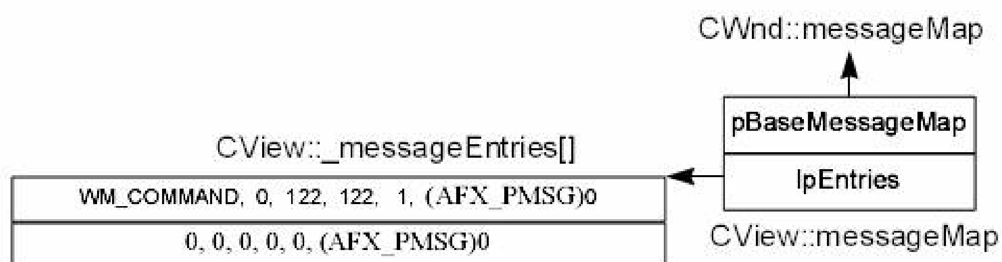
// in implementation file
#define CViewid 122
...
BEGIN_MESSAGE_MAP(CView, CWnd)
  ON_COMMAND(CViewid, 0)
END_MESSAGE_MAP()
```

就被展开成为：

```
// in header file
class CView : public CWnd
{
public:
  ...
  static AFX_MSGMAP_ENTRY _messageEntries[];
  static AFX_MSGMAP messageMap;
  virtual AFX_MSGMAP* GetMessageMap() const;
};

// in implementation file
AFX_MSGMAP* CView::GetMessageMap() const
       { return &CView::messageMap; }
AFX_MSGMAP CView::messageMap =
{ &(CWnd::messageMap),
       (AFX_MSGMAP_ENTRY*) &(CView::_messageEntries) };
AFX_MSGMAP_ENTRY CView::_messageEntries[] =
{
  { WM_COMMAND, 0, (WORD)122, (WORD)122, 1, (AFX_PMSG)0 },
  { 0, 0, 0, 0, 0, (AFX_PMSG)0 }
};
```

以图表示则为：



我们还可以定义各种类似 ON_COMMAND 这样的宏，把各式各样的消息与特定的处理例程关联起来。MFC 里头就有名为 ON_WM_PAINT、ON_WM_CREATE、ON_WM_SIZE...等等的宏。

我在 Frame7 范例程序中为 CCmdTarget 的每一派生类都产生类似上图的消息映射表：

```
// in header files
class CObject
{
    ... // 注意：CObject 并不属于消息流动网的一份子。
};
class CCmdTarget : public CObject
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CWinThread : public CCmdTarget
{
    ... // 注意：CWinThread 并不属于消息流动网的一份子。
};
class CWinApp : public CWinThread
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CDocument : public CCmdTarget
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CWnd : public CCmdTarget
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CFrameWnd : public CWnd
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CView : public CWnd
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CMyWinApp : public CWinApp
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CMyFrameWnd : public CFrameWnd
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CMyDoc : public CDocument
{
    ...
    DECLARE_MESSAGE_MAP()
};
class CMyView : public CView
{
    ...
    DECLARE_MESSAGE_MAP()
};
```

并且把各消息映射表的关联性架设起来，给予初值（每一个映射表都只有 ON_COMMAND一个项目）：

```
// in implementation files
BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
ON_COMMAND(CWndid, 0)
END_MESSAGE_MAP()
BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
ON_COMMAND(CFrameWndid, 0)
END_MESSAGE_MAP()
BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
ON_COMMAND(CDocumentid, 0)
END_MESSAGE_MAP()
BEGIN_MESSAGE_MAP(CView, CWnd)
ON_COMMAND(CViewid, 0)
END_MESSAGE_MAP()
BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
ON_COMMAND(CWinAppid, 0)
END_MESSAGE_MAP()
BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
ON_COMMAND(CMyWinAppid, 0)
END_MESSAGE_MAP()
BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
ON_COMMAND(CMyFrameWndid, 0)
END_MESSAGE_MAP()
BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
ON_COMMAND(CMyDocid, 0)
END_MESSAGE_MAP()
BEGIN_MESSAGE_MAP(CMyView, CView)
ON_COMMAND(CMyViewid, 0)
END_MESSAGE_MAP()
```

同时也设定了消息的终极镖靶 CCmdTarget 的映射表内容：

```
AFX_MSGMAP CCmdTarget::messageMap =
{
    NULL,
    &CCmdTarget::_messageEntries[0]
};
AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
{
    { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
};
```

于是，整个消息流动网就隐然成形了（图 3-5）。

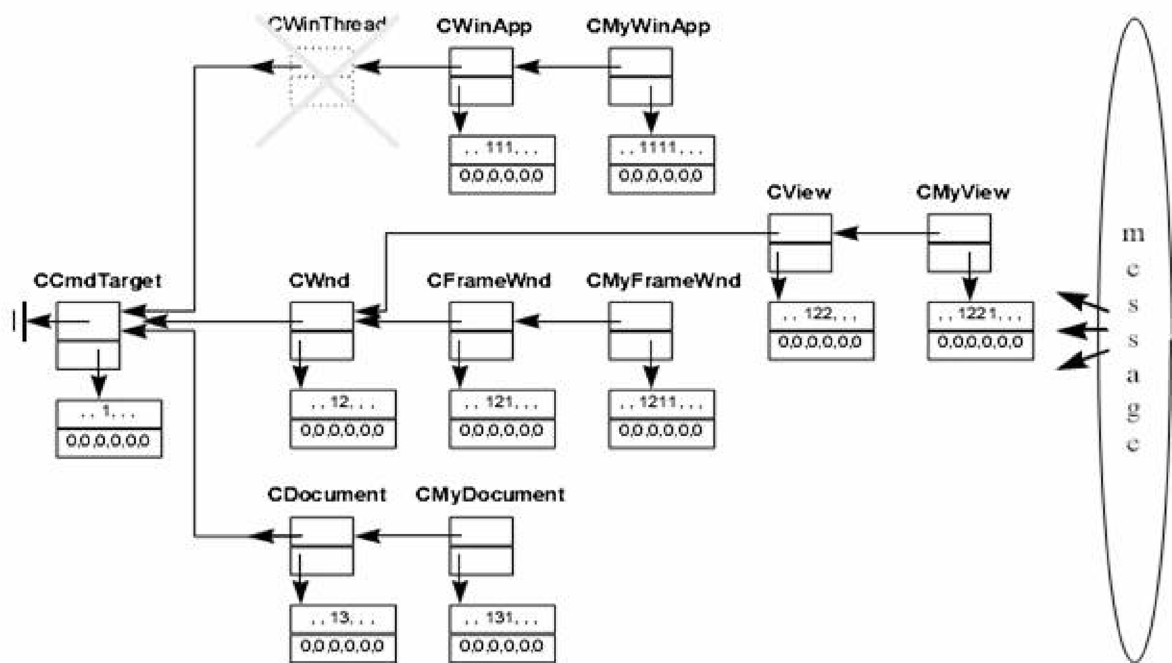


图3-5 Frame7程序所架构起来的消息流动网（也就是Message Map）

为了验证整个消息映射表，我必须在映射表中做点记号，等全部构造完成之后，再一一追踪把记号显示出来。我将为每一个类的消息映射表加上这个项目：

ON_COMMAND(Classid, 0)

这样就可以把 Classid 嵌到映射表中当作记号。正式用途（于 MFC 中）当然不是这样，这只不过是权宜之计。

在main函数中，我先产生四个对象（分别是CMyWinApp、CMyFrameWnd、CMyDoc、CMyView 对象）：

```
CMyWinApp theApp;  // theApp 是 CMyWinApp 物件
void main()
{
    CWinApp* pApp = AfxGetApp();
    pApp->InitApplication();
    pApp->InitInstance();   // 產生 CMyFrameWnd 物件
    pApp->Run();

     CMyDoc* pMyDoc = new CMyDoc;         // 产生 CMyDoc 对象
     CMyView* pMyView = new CMyView; // 产生 CMyView 对象
     CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
     ...
 }
```

然后分别取其消息映射表，一路追踪上去，把每一个消息映射表中的类记号打印出来：

```
void main()
{
...
   AFX_MSGMAP* pMessageMap = pMyView->GetMessageMap();
   cout << endl << "CMyView Message Map : " << endl;
   MsgMapPrinting(pMessageMap);

   pMessageMap = pMyDoc->GetMessageMap();
   cout << endl << "CMyDoc Message Map : " << endl;
   MsgMapPrinting(pMessageMap);

   pMessageMap = pMyFrame->GetMessageMap();
   cout << endl << "CMyFrameWnd Message Map : " << endl;
   MsgMapPrinting(pMessageMap);

   pMessageMap = pApp->GetMessageMap();
   cout << endl << "CMyWinApp Message Map : " << endl;
   MsgMapPrinting(pMessageMap);
}
```

下面这个函数追踪并打印消息映射表中的classid记号：

```
void MsgMapPrinting(AFX_MSGMAP* pMessageMap)
{
   for(; pMessageMap != NULL; pMessageMap = pMessageMap->pBaseMessageMap)
{
       AFX_MSGMAP_ENTRY* lpEntry = pMessageMap->lpEntries;
       printlpEntries(lpEntry);
   }
}

void  printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
{
struct {
  int classid;
  char* classname;
} classinfo[] = {
                    CCmdTargetid,  "CCmdTarget",
                    CWinThreadid,  "CWinThread",
                    CWinAppid,      "CWinApp",
                    CMyWinAppid,   "CMyWinApp",
                    CWndid,        "CWnd",
                    CFrameWndid,   "CFrameWnd",
                    CMyFrameWndid, "CMyFrameWnd",
                    CViewid,      "CView",
                    CMyViewid,    "CMyView",
                    CDocumentid, "CDocument",
                    CMyDocid,     "CMyDoc",
                    0,            "       "
               };

  for (int i=0; classinfo[i].classid != 0; i++)
  {
      if (classinfo[i].classid == lpEntry->nID)
      {
```

```
            cout << lpEntry->nID << "    ";
            cout << classinfo[i].classname << endl;
            break;
        }
    }
}
```

Frame7 的命令列编译链接动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

Frame7 的执行结果是：

```
CMyView Message Map :
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget

CMyDoc Message Map :
131    CMyDoc
13    CDocument
1    CCmdTarget

CMyFrameWnd Message Map :
1211    CMyFrameWnd
121    CFrameWnd
12    CWnd

1    CCmdTarget

CMyWinApp Message Map :
1111    CMyWinApp
111    CWinApp
1    CCmdTarget
```

Frame7 范例程序

MFC.H

```
#0001  #define TRUE 1
#0002  #define FALSE 0
#0003
#0004  typedef char* LPSTR;
#0005  typedef const char* LPCSTR;
#0006
#0007  typedef unsigned long  DWORD;
#0008  typedef int           BOOL;
#0009  typedef unsigned char  BYTE;
#0010  typedef unsigned short WORD;
#0011  typedef int            INT;
#0012  typedef unsigned int   UINT;
#0013  typedef long           LONG;
#0014
#0015  #define WM_COMMAND             0x0111
#0016  #define CObjectid              0xffff
#0017  #define   CCmdTargetid         1
#0018  #define     CWinThreadid       11
#0019  #define       CWinAppid        111
#0020  #define         CMyWinAppid    1111
#0021  #define     CWndid             12
#0022  #define       CFrameWndid      121
#0023  #define         CMyFrameWndid  1211
#0024  #define       CViewid          122
#0025  #define         CMyViewid      1221
#0026  #define     CDocumentid        13
#0027  #define       CMyDocid         131
#0028
#0029  #include <iostream.h>
#0030
#0031  ////////////////////////////////////////////////////////////////////////
#0032  // Window message map handling
#0033
```

```
#0034  struct AFX_MSGMAP_ENTRY;       // declared below after CWnd
#0035
#0036  struct AFX_MSGMAP
#0037  {
#0038          AFX_MSGMAP* pBaseMessageMap;
#0039          AFX_MSGMAP_ENTRY* lpEntries;
#0040  };
#0041
#0042  #define DECLARE_MESSAGE_MAP() \
#0043          static AFX_MSGMAP_ENTRY _messageEntries[]; \
#0044          static AFX_MSGMAP messageMap; \
#0045          virtual AFX_MSGMAP* GetMessageMap() const;
#0046
#0047  #define BEGIN_MESSAGE_MAP(theClass, baseClass) \
#0048          AFX_MSGMAP* theClass::GetMessageMap() const \
#0049                  { return &theClass::messageMap; } \
#0050          AFX_MSGMAP theClass::messageMap = \
#0051          { &(baseClass::messageMap), \
#0052                  (AFX_MSGMAP_ENTRY*) &(theClass::_messageEntries) }; \
#0053          AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
#0054          {
#0055
#0056  #define END_MESSAGE_MAP() \
#0057          { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
#0058          };
#0059
#0060  // Message map signature values and macros in separate header
#0061  #include "afxmsg_.h"
#0062
#0063  class CObject
#0064  {
#0065  public:
#0066    CObject::CObject()  {
#0067                        }
#0068    CObject::~CObject() {
#0069                        }
#0070  };
#0071
#0072  class CCmdTarget : public CObject
#0073  {
#0074  public:
#0075    CCmdTarget::CCmdTarget()  {
#0076                              }
#0077    CCmdTarget::~CCmdTarget() {
#0078                              }
#0079    DECLARE_MESSAGE_MAP()       // base class - no {{ }} macros
```

```
#0080 };
#0081
#0082 typedef void (CCmdTarget::*AFX_PMSG)(void);
#0083
#0084 struct AFX_MSGMAP_ENTRY  // MFC 4.0
#0085 {
#0086     UINT nMessage; // windows message
#0087     UINT nCode;    // control code or WM_NOTIFY code
#0088     UINT nID;      // control ID (or 0 for windows messages)
#0089     UINT nLastID;  // used for entries specifying a range of control id's
#0090     UINT nSig;     // signature type (action) or pointer to message #
#0091     AFX_PMSG pfn;  // routine to call (or special value)
#0092 };
#0093
#0094 class CWinThread : public CCmdTarget
#0095 {
#0096 public:
#0097   CWinThread::CWinThread()  {
#0098                             }
#0099   CWinThread::~CWinThread() {
#0100                             }
#0101
#0102   virtual BOOL InitInstance() {
#0103                                 cout << "CWinThread::InitInstance \n";
#0104                                 return TRUE;
#0105                               }
#0106   virtual int Run() {
#0107                       cout << "CWinThread::Run \n";
#0108                       return 1;
#0109                     }
#0110 };
#0111
#0112 class CWnd;
#0113
#0114 class CWinApp : public CWinThread
#0115 {
#0116 public:
#0117   CWinApp* m_pCurrentWinApp;
#0118   CWnd* m_pMainWnd;
#0119
#0120 public:
#0121   CWinApp::CWinApp()  {
#0122                         m_pCurrentWinApp = this;
#0123                       }
#0124   CWinApp::~CWinApp() {
#0125                       }
#0126
#0127   virtual BOOL InitApplication() {
#0128                                    cout << "CWinApp::InitApplication \n";
#0129                                    return TRUE;
#0130                                  }
#0131   virtual BOOL InitInstance()  {
#0132                                  cout << "CWinApp::InitInstance \n";
#0133                                  return TRUE;
```

```
#0134                                  }
#0135   virtual int Run() {
#0136                       cout << "CWinApp::Run \n";
#0137                       return CWinThread::Run();
#0138                     }
#0139
#0140   DECLARE_MESSAGE_MAP()
#0141 };
#0142
#0143 typedef void (CWnd::*AFX_PMSGW)(void);
#0144         // like 'AFX_PMSG' but for CWnd derived classes only
#0145
#0146 class CDocument : public CCmdTarget
#0147 {
#0148 public:
#0149   CDocument::CDocument()   {
#0150                            }
#0151   CDocument::~CDocument()  {
#0152                            }
#0153   DECLARE_MESSAGE_MAP()
#0154 };
#0155
#0156 class CWnd : public CCmdTarget
#0157 {
#0158 public:
#0159   CWnd::CWnd()   {
#0160                  }
#0161   CWnd::~CWnd()  {
#0162                  }
#0163
#0164   virtual BOOL Create();
#0165   BOOL CreateEx();
#0166   virtual BOOL PreCreateWindow();
#0167
#0168   DECLARE_MESSAGE_MAP()
#0169 };
#0170
#0171 class CFrameWnd : public CWnd
#0172 {
#0173 public:
#0174   CFrameWnd::CFrameWnd()   {
#0175                            }
#0176   CFrameWnd::~CFrameWnd()  {
#0177                            }
#0178   BOOL Create();
#0179   virtual BOOL PreCreateWindow();
#0180
#0181   DECLARE_MESSAGE_MAP()
#0182 };
#0183
#0184 class CView : public CWnd
#0185 {
#0186 public:
```

```
#0187   CView::CView()   {
#0188                    }
#0189   CView::~CView()  {
#0190                    }
#0191   DECLARE_MESSAGE_MAP()
#0192 };
#0193
#0194 // global function
#0195 CWinApp* AfxGetApp();
```

AFXMSG_.H

```
#0001 enum AfxSig
#0002 {
#0003        AfxSig_end = 0,     // [marks end of message map]
#0004        AfxSig_vv,
#0005 };
#0006
#0007 #define ON_COMMAND(id, memberFxn) \
#0008     { WM_COMMAND, 0, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },
```

MFC.CPP

```
#0001 #include "my.h"  // 原该含入 mfc.h 就好，但为了 CMyWinApp 的定义，所以...
#0002
#0003 extern CMyWinApp theApp;
#0004
#0005 BOOL CWnd::Create()
#0006 {
#0007   cout << "CWnd::Create \n";
```

```
#0008    return TRUE;
#0009  }
#0010
#0011  BOOL CWnd::CreateEx()
#0012  {
#0013    cout << "CWnd::CreateEx \n";
#0014    PreCreateWindow();
#0015    return TRUE;
#0016  }
#0017
#0018  BOOL CWnd::PreCreateWindow()
#0019  {
#0020    cout << "CWnd::PreCreateWindow \n";
#0021    return TRUE;
#0022  }
#0023
#0024  BOOL CFrameWnd::Create()
#0025  {
#0026    cout << "CFrameWnd::Create \n";
#0027    CreateEx();
#0028    return TRUE;
#0029  }
#0030
#0031  BOOL CFrameWnd::PreCreateWindow()
#0032  {
#0033    cout << "CFrameWnd::PreCreateWindow \n";
#0034    return TRUE;
#0035  }
```

```
#0036
#0037  CWinApp* AfxGetApp()
#0038  {
#0039    return theApp.m_pCurrentWinApp;
#0040  }
#0041
#0042  AFX_MSGMAP* CCmdTarget::GetMessageMap() const
#0043  {
#0044          return &CCmdTarget::messageMap;
#0045  }
#0046
#0047  AFX_MSGMAP CCmdTarget::messageMap =
#0048  {
#0049          NULL,
#0050          &CCmdTarget::_messageEntries[0]
#0051  };
#0052
#0053  AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
```

```
#0054  {
#0055      // { 0, 0, 0, 0, AfxSig_end, 0 }     // nothing here
#0056      { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
#0057
#0058  };
#0059
#0060  BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
#0061  ON_COMMAND(CWndid, 0)
#0062  END_MESSAGE_MAP()
#0063
#0064  BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
#0065  ON_COMMAND(CFrameWndid, 0)
#0066  END_MESSAGE_MAP()
#0067
#0068  BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
#0069  ON_COMMAND(CDocumentid, 0)
#0070  END_MESSAGE_MAP()
#0071
#0072  BEGIN_MESSAGE_MAP(CView, CWnd)
#0073  ON_COMMAND(CViewid, 0)
#0074  END_MESSAGE_MAP()
#0075
#0076  BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
#0077  ON_COMMAND(CWinAppid, 0)
#0078  END_MESSAGE_MAP()
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()    {
#0008                              }
#0009    CMyWinApp::~CMyWinApp()   {
#0010                              }
#0011
#0012    virtual BOOL InitInstance();
#0013    DECLARE_MESSAGE_MAP()
#0014  };
#0015
#0016  class CMyFrameWnd : public CFrameWnd
#0017  {
#0018  public:
```

```
#0019   CMyFrameWnd();
#0020   ~CMyFrameWnd()  {
#0021                   }
#0022   DECLARE_MESSAGE_MAP()
#0023 };
#0024
#0025 class CMyDoc : public CDocument
#0026 {
#0027 public:
#0028   CMyDoc::CMyDoc()  {
#0029                     }
#0030   CMyDoc::~CMyDoc() {
#0031                     }
#0032   DECLARE_MESSAGE_MAP()
#0033 };
#0034
#0035 class CMyView : public CView
#0036 {
#0037 public:
#0038   CMyView::CMyView()   {
#0039                        }
#0040   CMyView::~CMyView()  {
#0041                        }
#0042   DECLARE_MESSAGE_MAP()
#0043 };
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp;
#0004
#0005 BOOL CMyWinApp::InitInstance()
#0006 {
#0007     cout << "CMyWinApp::InitInstance \n";
#0008     m_pMainWnd = new CMyFrameWnd;
#0009     return TRUE;
#0010 }
#0011
#0012 CMyFrameWnd::CMyFrameWnd()
#0013 {
#0014     Create();
#0015 }
#0016
#0017 BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
#0018 ON_COMMAND(CMyWinAppid, 0)
#0019 END_MESSAGE_MAP()
```

```
#0020
#0021  BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0022  ON_COMMAND(CMyFrameWndid, 0)
#0023  END_MESSAGE_MAP()
#0024
#0025  BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
#0026  ON_COMMAND(CMyDocid, 0)
#0027  END_MESSAGE_MAP()
#0028
#0029  BEGIN_MESSAGE_MAP(CMyView, CView)
#0030  ON_COMMAND(CMyViewid, 0)
#0031  END_MESSAGE_MAP()
#0032
#0033  void  printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
#0034  {
#0035  struct {
#0036    int classid;
#0037    char* classname;
#0038  } classinfo[] = {
#0039                       CCmdTargetid,  "CCmdTarget",
#0040                       CWinThreadid,  "CWinThread",
#0041                       CWinAppid,      "CWinApp",
#0042                       CMyWinAppid,    "CMyWinApp",
#0043                       CWndid,         "CWnd",
#0044                       CFrameWndid,    "CFrameWnd",
#0045                       CMyFrameWndid,  "CMyFrameWnd",
#0046                       CViewid,         "CView",
#0047                       CMyViewid,    "CMyView",
#0048                       CDocumentid,  "CDocument",
#0049                       CMyDocid,     "CMyDoc",
#0050                       0,            "     "
#0051                   };
#0052
#0053    for (int i=0; classinfo[i].classid != 0; i++)
#0054    {
#0055        if (classinfo[i].classid == lpEntry->nID)
#0056        {
#0057            cout << lpEntry->nID << "    ";
#0058            cout << classinfo[i].classname << endl;
#0059            break;
#0060        }
#0061    }
#0062  }
#0063
#0064  void MsgMapPrinting(AFX_MSGMAP* pMessageMap)
#0065  {
#0066      for(; pMessageMap != NULL; pMessageMap = pMessageMap->pBaseMessageMap) {
#0067          AFX_MSGMAP_ENTRY* lpEntry = pMessageMap->lpEntries;
#0068          printlpEntries(lpEntry);
#0069      }
#0070  }
#0071
#0072  //------------------------------------------------------------------
#0073  // main
#0074  //------------------------------------------------------------------
```

```
#0075  void main()
#0076  {
#0077
#0078      CWinApp* pApp = AfxGetApp();
#0079
#0080      pApp->InitApplication();
#0081      pApp->InitInstance();
#0082      pApp->Run();
#0083
#0084      CMyDoc* pMyDoc = new CMyDoc;
#0085      CMyView* pMyView = new CMyView;
#0086      CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
#0087
#0088      // output Message Map construction
#0089      AFX_MSGMAP* pMessageMap = pMyView->GetMessageMap();
#0090      cout << endl << "CMyView Message Map : " << endl;
#0091      MsgMapPrinting(pMessageMap);
#0092
#0093      pMessageMap = pMyDoc->GetMessageMap();
#0094      cout << endl << "CMyDoc Message Map : " << endl;
#0095      MsgMapPrinting(pMessageMap);
#0096
#0097      pMessageMap = pMyFrame->GetMessageMap();
#0098      cout << endl << "CMyFrameWnd Message Map : " << endl;
#0099      MsgMapPrinting(pMessageMap);
#0100
#0101      pMessageMap = pApp->GetMessageMap();
#0102      cout << endl << "CMyWinApp Message Map : " << endl;
#0103      MsgMapPrinting(pMessageMap);
#0104  }
```

Command Routing（命令循环）

我们已经在上一节把整个消息流动网架设起来了。当消息进来，会有一个捕获推动它前进。消息如何进来，以及捕获函数如何推动，都是属于Windows程序设计的范畴，暂时不管。我现在要仿真出消息的流动循环路线 -- 我常喜欢称之为消息的「二万五千里长征」。

消息如果是从子类流向父类（纵向流动），那么事情再简单不过，整个 Message Map消息映射表已规划出十分明确的路线。但是正如上一节一开始我说的，MFC之中用来处理消息的C++ 类并不呈单鞭发展，作为application framework 的重要架构之一的document/view，也具有处理消息的能力（你现在可能还不清楚什么是document/view，没有关系）；因此，消息应该有横向流动的机会。MFC 对于消息循环的规定是：

- 如果是一般的Windows消息（WM_xxx），一定是由派生类流向基类，没有旁流的可能。
- 如果是命令消息WM_COMMAND，就有奇特的路线了：

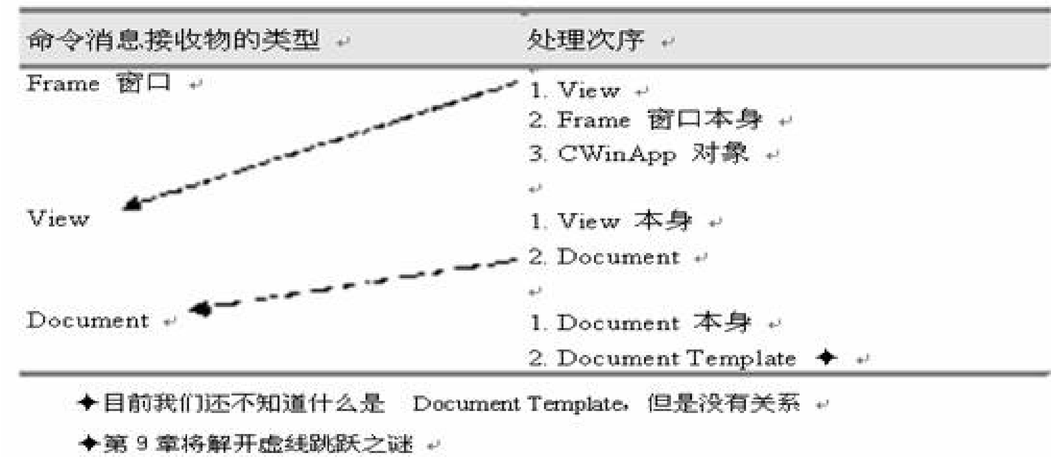


图3-6 MFC对于命令消息WM_COMMAND的特殊处理顺序

不管这个规则是怎么定下来的，现在我要设计一个推动引擎，把它仿真出来。以下这些函数名称以及函数内容，完全模拟 MFC 内部。有些函数似乎赘余，那是因为我删掉了许多主题以外的动作。不把看似赘余的函数拿掉或合并，是为了留下 MFC 的足迹。此外，为了追踪调用过程（call stack），我在各函数的第一行输出一串识别文字。

首先我把新增加的一些成员函数做个列表：

类	与消息循环有关的成员函数	注意
none	*AfxWndProc*	global
none	*AfxCallWndProc*	global
CCmdTarget	*OnCmdMsg*	virtual
CDocument	*OnCmdMsg*	virtual
CWnd	*WindowProc*	virtual
	OnCommand	virtual
	DefWindowProc	virtual
CFrameWnd	*OnCommand*	virtual
	OnCmdMsg	virtual
CView	*OnCmdMsg*	virtual

全局函数AfxWndProc就是我所谓的推动引擎的起始点 。它本来应该是在CWinThread::Run中被调用，但为了实验目的，我在main中调用它，每调用一次便推送一个消息。这个函数在 MFC 中有四个参数，为了方便，我加上第五个，用以表示是谁获得消息（成为循环的起点）。例如：

```
AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
```

表示pMyFrame获得了一个WM_CREATE，而：

```
AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
```

表示pMyView 获得了一个WM_COMMAND。

下面是消息流动的过程：

```
LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
                   CWnd *pWnd)  // last param. pWnd is added by JJHou.
{
  cout << "AfxWndProc()" << endl;
  return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
}

LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
                       WPARAM wParam, LPARAM lParam)
{
  cout << "AfxCallWndProc()" << endl;
  LRESULT lResult = pWnd->WindowProc(nMsg, wParam, lParam);
  return lResult;
}
```

pWnd->WindowProc 究竟是调用哪一个函数？不一定，得视 pWnd 到底指向何种类之对象而定 -- 别忘了 WindowProc 是虚函数。这正是虚函数发挥它功效的地方呀：

如果pWnd指向CMyFrameWnd对象，那么调用的是CFrameWnd::WindowProc。而 因 为 CFrameWnd并没有改写WindowProc ，所以调用的其实是CWnd::WindowProc。

如果pWnd 指向CMyView对象，那么调用的是CView::WindowProc。而因为 CView并没有改写 WindowProc，所以调用的其实是 CWnd::WindowProc。虽然殊途同归，意义上是不相同的。切记！切记！

CWnd::WindowProc 首先判断消息是否为 WM_COMMAND。如果不是，事情最单纯，就把消息往父类推去，父类再往祖父类推去。每到一个类的消息映射表，原本应该比对 AFX_MSGMAP_ENTRY 的每一个元素，比对成功就调用对应的处理例程。不过在这里我不作比对，只是把 AFX_MSGMAP_ENTRY 中的类识别代码印出来（就像上一节的 Frame7 程序一样），以表示「到此一游」：

```
LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    AFX_MSGMAP* pMessageMap;
    AFX_MSGMAP_ENTRY* lpEntry;

    if (nMsg == WM_COMMAND) // special case for commands
    {
        if (OnCommand(wParam, lParam))   ❶
            return 1L; // command handled
        else
            return (LRESULT)DefWindowProc(nMsg, wParam, lParam);   ❾
    }

    pMessageMap = GetMessageMap();

    for (; pMessageMap != NULL;
         pMessageMap = pMessageMap->pBaseMessageMap)
    {
            lpEntry = pMessageMap->lpEntries;
            printlpEntries(lpEntry);
    }
    return 0;  // J.J.Hou: if find, should call lpEntry->pfn,
               // otherwise should call DefWindowProc.
               // for simplification, we just return 0.
}
```

如果消息是 WM_COMMAND，CWnd::WindowProc 调用①OnCommand。好，注意了，这又是一个CWnd 的虚函数：

1.如果this指向CMyFrameWnd对象，那么调用的是CFrameWnd::OnCommand。

2. 如果this指向CMyView对象，那么调用的是CView::OnCommand。而因为CView并没有改写OnCommand，所以调用的其实是CWnd::OnCommand。

这次可就没有殊途同归了。

我们以第一种情况为例，再往下看：

```
BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    cout << "CFrameWnd::OnCommand()" << endl;
    // ...
    // route as normal command
    return CWnd::OnCommand(wParam, lParam);   ❷
}

BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    cout << "CWnd::OnCommand()" << endl;
    // ...
    return OnCmdMsg(0, 0);   ❸
}
```

又一次遭遇虚函数。经过前两次的分析，相信你对此很有经验了。③OnCmdMsg 是 CCmdTarget 的虚函数，所以：

1. 如果this指向CMyFrameWnd对象，那么调用的是CFrameWnd::OnCmdMsg。

2. 如果this指向CMyView对象，那么调用的是CView::OnCmdMsg。

3. 如果this 指向CMyDoc 对象，那么调用的是CDocument::OnCmdMsg。

4. 如果this指向CMyWinApp 对象，那么调用的是CWinApp::OnCmdMsg。而因为CWinApp并没有改写OnCmdMsg，所以调用的其实是 CCmdTarget::OnCmdMsg。

目前的情况是第一种，于是调用 CFrameWnd::OnCmdMsg：

```
BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CFrameWnd::OnCmdMsg()" << endl;
    // pump through current view FIRST
    CView* pView = GetActiveView();
    if (pView->OnCmdMsg(nID, nCode))    ❹
            return TRUE;

    // then pump through frame
    if (CWnd::OnCmdMsg(nID, nCode))    ❼
            return TRUE;

    // last but not least, pump through app
    CWinApp* pApp = AfxGetApp();

    if (pApp->OnCmdMsg(nID, nCode))    ❽
            return TRUE;

    return FALSE;
}
```

这个函数反应出图3-6Frame窗口处理WM_COMMAND 的次序。最先调用的是④pView->OnCmdMsg，于是：

```
BOOL CView::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CView::OnCmdMsg()" << endl;
    if (CWnd::OnCmdMsg(nID, nCode))    ❺
        return TRUE;

    BOOL bHandled = FALSE;
    bHandled = m_pDocument->OnCmdMsg(nID, nCode);    ❻
    return bHandled;
}
```

这又反应出图3-6 View窗口处理WM_COMMAND的次序。最先调用的是⑤Wnd::OnCmdMsg，而CWnd并未改写OnCmdMsg， 所以其实就是调用CmdTarget::OnCmdMsg：

```
BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CCmdTarget::OnCmdMsg()" << endl;
    // Now look through message map to see if it applies to us
    AFX_MSGMAP* pMessageMap;
    AFX_MSGMAP_ENTRY* lpEntry;
    for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
         pMessageMap = pMessageMap->pBaseMessageMap)
    {
           lpEntry = pMessageMap->lpEntries;
           printlpEntries(lpEntry);
    }

    return FALSE;   // not handled
}
```

这是一个走访消息映射表的动作。注意，GetMessageMap 也是个虚函数（隐藏在DECLARE_MESSAGE_MAP 宏定义中），所以它所得到的消息映射表将是 this（以目前而言是pMyView）所指对象的映射表。于是我们得到了这个结果：

```
pMyFrame received a WM_COMMAND, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CFrameWnd::GetActiveView()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget
```

如果在映射表中找到了对应的消息，就调用对应的处理例程，然后也就结束了二万五千里长征。如果没找到，长征还没有结束，这时候退守回到CView::OnCmdMsg，调用⑥CDocument::OnCmdMsg：

```
BOOL CDocument::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CDocument::OnCmdMsg()" << endl;
    if (CCmdTarget::OnCmdMsg(nID, nCode))
        return TRUE;

    return FALSE;
}
```

于是得到这个结果：

```
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
131    CMyDoc
13    CDocument
1    CCmdTarget
```

如果在映射表中还是没找到对应消息，二万五千里长征还是未能结束，这时候退守回到CFrameWnd::OnCmdMsg，调用⑦CWnd::OnCmdMsg（也就是CCmdTarget::OnCmdMsg），得到这个结果：

```
CCmdTarget::OnCmdMsg()
1211    CMyFrameWnd
121    CFrameWnd
12    CWnd
1    CCmdTarget
```

如果在映射表中还是没找到对应消息，二万五千里长征还是未能结束，再退回到CFrameWnd::OnCmdMsg，调用⑧CWinApp::OnCmdMsg（亦即CCmdTarget::OnCmdMsg），得到这个结果：

```
1111    CMyWinApp
111    CWinApp
1    CCmdTarget
```

万一还是没找到对应的消息， 二万五千里长征可也穷途末路了，退回到CWnd::WindowProc，调用 ⑨CWnd::DefWindowProc。你可以想象，在真正的MFC中这个成员函数必是调用Windows API函数::DefWindowProc。为了简化，我让它在Frame8 中是个空函数。

故事结束！

我以图 3-7 表示这二万五千里长征的调用次序（call stack），图 3-8 表示这二万五千里长征的消息流动路线。

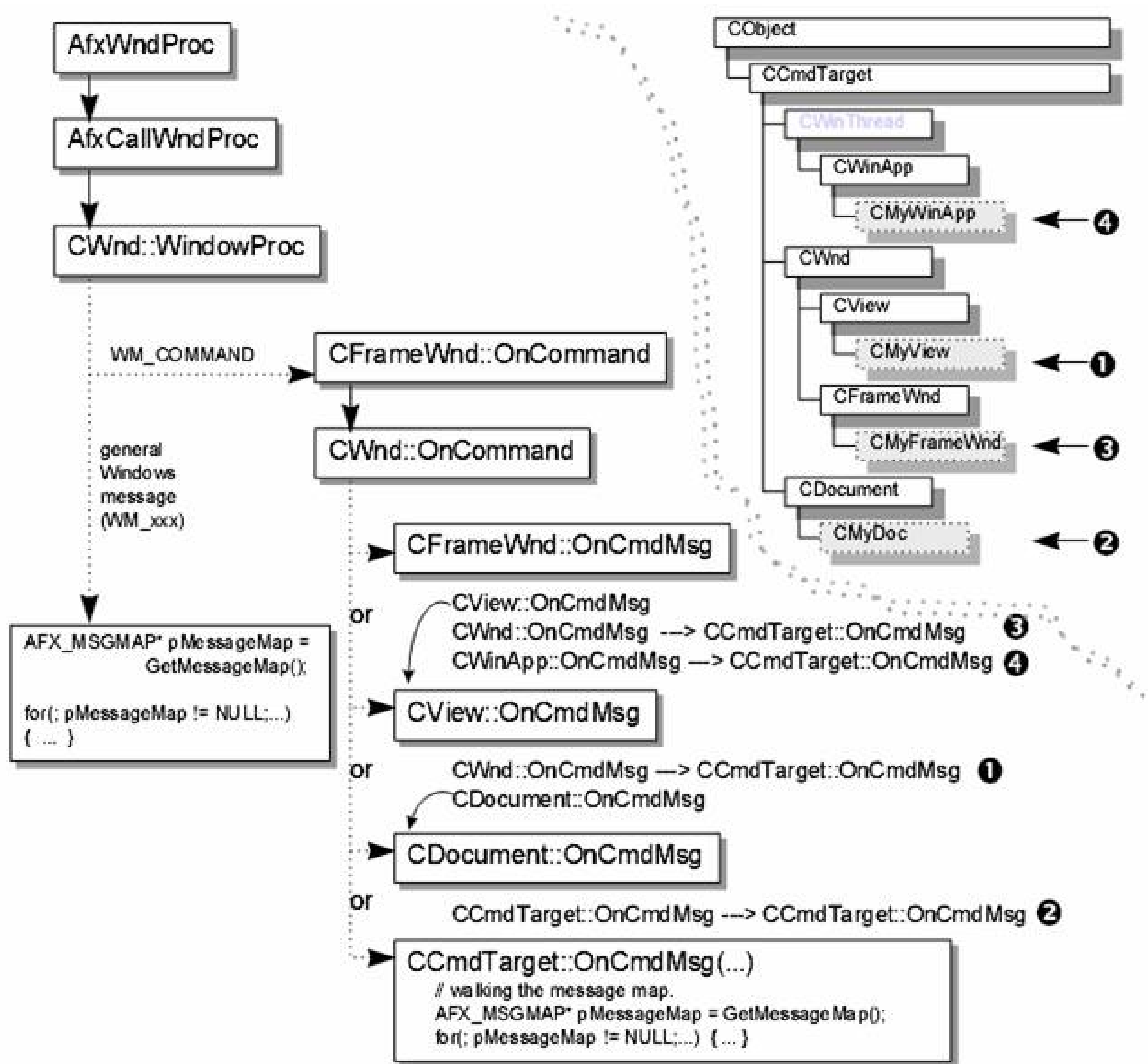


图3-7 当CMyFrameWnd对象获得一个WM_COMMAND，

所引起的Frame8函数调用次序

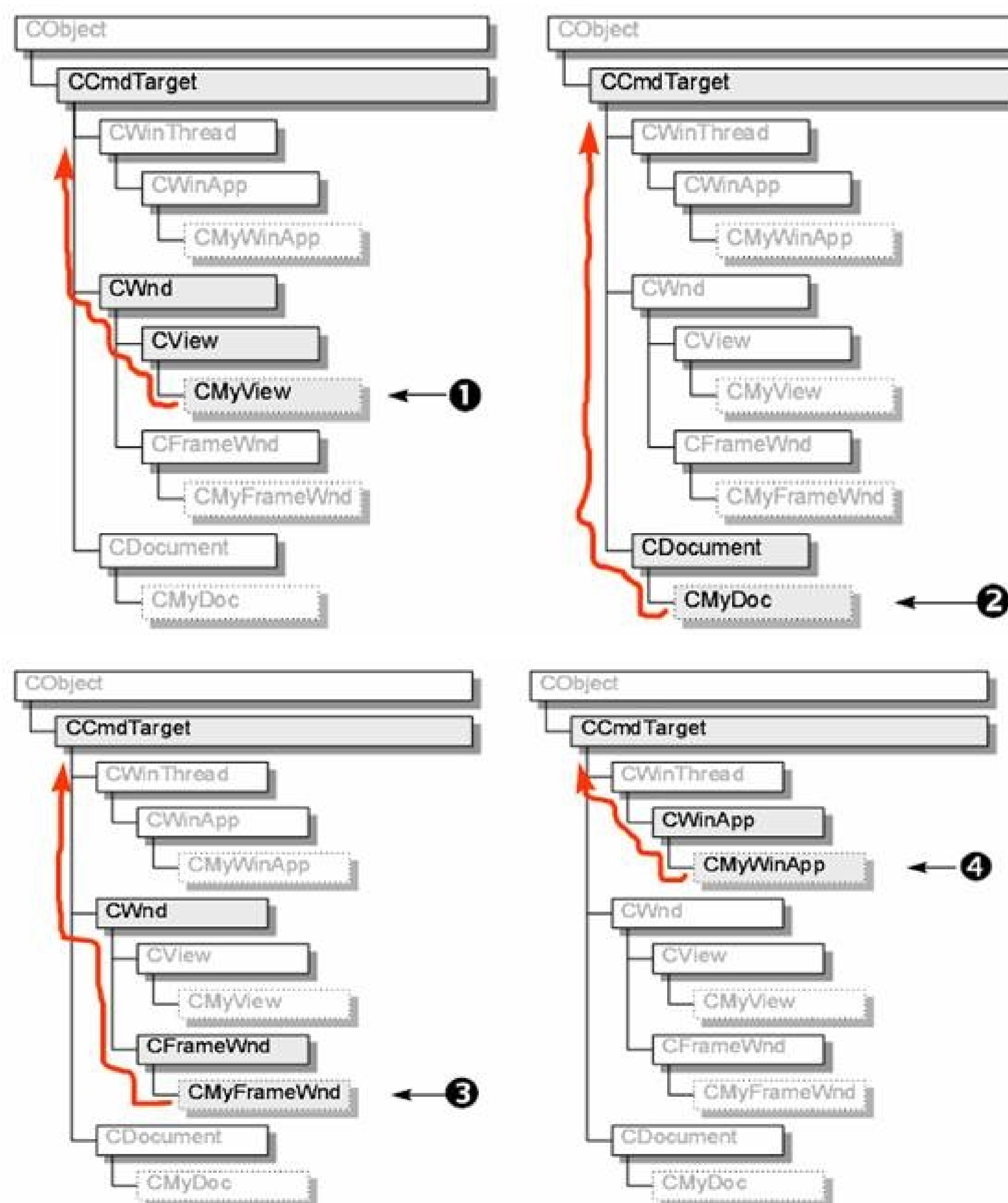


图3-8 当CMyFrameWnd对象获得一个WM_COMMAND，

所引起的消息流动路线

Frame8 测试四种情况：分别从 frame 对象和 view 对象中推动消息，消息分一般Windows 消息和 WM_COMMAND 两种：

```
// test Message Routing
AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
AfxWndProc(0, WM_PAINT, 0, 0, pMyView);
AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
AfxWndProc(0, WM_COMMAND, 0, 0, pMyFrame);
```

Frame8 的命令列编译链接动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

以下是 Frame8 的执行结果：

```
CWinApp::InitApplication
CMyWinApp::InitInstance
CFrameWnd::Create
CWnd::CreateEx
CFrameWnd::PreCreateWindow
CWinApp::Run
CWinThread::Run

pMyFrame received a WM_CREATE, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
1211    CMyFrameWnd
121    CFrameWnd
12    CWnd
1    CCmdTarget

pMyView received a WM_PAINT, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget

pMyView received a WM_COMMAND, routing path and call stack:
AfxWndProc()
```

```
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnCommand()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
131    CMyDoc
13    CDocument
1    CCmdTarget
CWnd::DefWindowProc()
```

```
pMyFrame received a WM_COMMAND, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CFrameWnd::GetActiveView()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
131    CMyDoc
13    CDocument
1    CCmdTarget
CCmdTarget::OnCmdMsg()
1211    CMyFrameWnd
121    CFrameWnd
12    CWnd
1    CCmdTarget
CCmdTarget::OnCmdMsg()
1111    CMyWinApp
111    CWinApp
1    CCmdTarget
CWnd::DefWindowProc()
```

Frame8 范例程序

MFC.H

```
#0001  #define TRUE 1
#0002  #define FALSE 0
#0003
#0004  typedef char* LPSTR;
#0005  typedef const char* LPCSTR;
#0006
#0007  typedef unsigned long  DWORD;
#0008  typedef int            BOOL;
```

```
#0009  typedef unsigned char  BYTE;
#0010  typedef unsigned short WORD;
#0011  typedef int            INT;
#0012  typedef unsigned int   UINT;
#0013  typedef long           LONG;
#0014
#0015  typedef UINT          WPARAM;
#0016  typedef LONG          LPARAM;
#0017  typedef LONG          LRESULT;
#0018  typedef int           HWND;
#0019
#0020  #define WM_COMMAND              0x0111
#0021  #define WM_CREATE               0x0001
#0022  #define WM_PAINT                0x000F
#0023  #define WM_NOTIFY               0x004E
#0024
#0025  #define CObjectid               0xffff
#0026  #define   CCmdTargetid           1
#0027  #define     CWinThreadid         11
#0028  #define       CWinAppid          111
#0029  #define         CMyWinAppid      1111
#0030  #define     CWndid               12
#0031  #define       CFrameWndid        121
#0032  #define         CMyFrameWndid  1211
#0033  #define       CViewid          122
#0034  #define         CMyViewid      1221
#0035  #define     CDocumentid        13
#0036  #define       CMyDocid         131
#0037
#0038  #include <iostream.h>
#0039
#0040  ///////////////////////////////////////////////////////
#0041  // Window message map handling
#0042
#0043  struct AFX_MSGMAP_ENTRY;       // declared below after CWnd
#0044
#0045  struct AFX_MSGMAP
#0046  {
#0047          AFX_MSGMAP* pBaseMessageMap;
#0048          AFX_MSGMAP_ENTRY* lpEntries;
#0049  };
#0050
#0051  #define DECLARE_MESSAGE_MAP() \
#0052          static AFX_MSGMAP_ENTRY _messageEntries[]; \
#0053          static AFX_MSGMAP messageMap; \
#0054          virtual AFX_MSGMAP* GetMessageMap() const;
#0055
#0056  #define BEGIN_MESSAGE_MAP(theClass, baseClass) \
#0057          AFX_MSGMAP* theClass::GetMessageMap() const \
#0058                  { return &theClass::messageMap; } \
#0059          AFX_MSGMAP theClass::messageMap = \
#0060          { &(baseClass::messageMap), \
```

```
#0061                (AFX_MSGMAP_ENTRY*) &(theClass::_messageEntries) }; \
#0062         AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
#0063         {
#0064
#0065 #define END_MESSAGE_MAP() \
#0066         { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
#0067         };
#0068
#0069 // Message map signature values and macros in separate header
#0070 #include "afxmsg_.h"
#0071
#0072 class CObject
#0073 {
#0074 public:
#0075   CObject::CObject()  {
#0076                       }
#0077   CObject::~CObject() {
#0078                       }
#0079 };
#0080
#0081 class CCmdTarget : public CObject
#0082 {
#0083 public:
#0084   CCmdTarget::CCmdTarget()  {
#0085                             }
#0086   CCmdTarget::~CCmdTarget() {

#0087                             }
#0088
#0089   virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0090
#0091   DECLARE_MESSAGE_MAP()       // base class - no {{ }} macros
#0092 };
#0093
#0094 typedef void (CCmdTarget::*AFX_PMSG)(void);
#0095
#0096 struct AFX_MSGMAP_ENTRY  // MFC 4.0
#0097 {
#0098     UINT nMessage; // windows message
#0099     UINT nCode;    // control code or WM_NOTIFY code
#0100     UINT nID;      // control ID (or 0 for windows messages)

#0101     UINT nLastID;  // used for entries specifying a range of control id's
#0102     UINT nSig;     // signature type (action) or pointer to message #
#0103     AFX_PMSG pfn;  // routine to call (or special value)
#0104 };
#0105
#0106 class CWinThread : public CCmdTarget
#0107 {
#0108 public:
#0109   CWinThread::CWinThread()  {
#0110                             }
#0111   CWinThread::~CWinThread() {
#0112                             }
#0113
#0114   virtual BOOL InitInstance() {
#0115                                 cout << "CWinThread::InitInstance \n";
#0116                                 return TRUE;
```

```
#0117                                    }
#0118   virtual int Run() {
#0119                               cout << "CWinThread::Run \n";
#0120                               // AfxWndProc(...);
#0121                               return 1;
#0122                             }
#0123  };
#0124
#0125  class CWnd;
#0126
#0127  class CWinApp : public CWinThread
#0128  {
#0129  public:
#0130    CWinApp* m_pCurrentWinApp;
#0131    CWnd* m_pMainWnd;
#0132

#0133  public:
#0134    CWinApp::CWinApp()  {
#0135                          m_pCurrentWinApp = this;
#0136                        }
#0137    CWinApp::~CWinApp() {
#0138                        }
#0139
#0140    virtual BOOL InitApplication() {
#0141                                      cout << "CWinApp::InitApplication \n";
#0142                                      return TRUE;
#0143                                     }
#0144    virtual BOOL InitInstance()   {
#0145                                     cout << "CWinApp::InitInstance \n";
#0146                                     return TRUE;
#0147                                   }
#0148    virtual int Run() {
#0149                        cout << "CWinApp::Run \n";
#0150                        return CWinThread::Run();

#0151                         }
#0152
#0153    DECLARE_MESSAGE_MAP()
#0154  };
#0155
#0156  typedef void (CWnd::*AFX_PMSGW)(void);
#0157                // like 'AFX_PMSG' but for CWnd derived classes only
#0158
#0159  class CDocument : public CCmdTarget
#0160  {
#0161  public:
#0162    CDocument::CDocument()   {
#0163                             }
#0164    CDocument::~CDocument()  {
#0165                             }
#0166
#0167    virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0168
#0169    DECLARE_MESSAGE_MAP()
#0170  };
#0171
#0172  class CWnd : public CCmdTarget
```

```
#0173 {
#0174 public:
#0175   CWnd::CWnd()   {
#0176                  }
#0177   CWnd::~CWnd()  {
#0178                  }

#0179
#0180   virtual BOOL Create();
#0181   BOOL CreateEx();
#0182   virtual BOOL PreCreateWindow();
#0183   virtual LRESULT WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam);
#0184   virtual LRESULT DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam);
#0185   virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
#0186
#0187   DECLARE_MESSAGE_MAP()
#0188 };
#0189
#0190 class CView;
#0191
#0192 class CFrameWnd : public CWnd
#0193 {
#0194 public:
#0195   CView* m_pViewActive;      // current active view
#0196
#0197 public:
#0198   CFrameWnd::CFrameWnd()   {
#0199                            }
#0200   CFrameWnd::~CFrameWnd()  {
#0201                            }
#0202   BOOL Create();
#0203   CView* GetActiveView() const;
#0204   virtual BOOL PreCreateWindow();
#0205   virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
#0206   virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0207
#0208   DECLARE_MESSAGE_MAP()
#0209
#0210   friend CView;
```

```
#0211 };
#0212
#0213 class CView : public CWnd
#0214 {
#0215 public:
#0216   CDocument* m_pDocument;
#0217
#0218 public:
#0219   CView::CView()   {
#0220                    }
#0221   CView::~CView()  {
#0222                    }
#0223
#0224   virtual BOOL OnCmdMsg(UINT nID, int nCode);
```

```
#0225
#0226   DECLARE_MESSAGE_MAP()
#0227
#0228   friend CFrameWnd;
```

```
#0229 };
#0230
#0231 // global function
#0232 CWinApp* AfxGetApp();
#0233 LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
#0234                   CWnd* pWnd); // last param. pWnd is added by JJHOU.
#0235 LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg, WPARAM wParam,
#0236                       LPARAM lParam);
```

AFXMSG_.H

```
#0001 enum AfxSig
#0002 {
#0003        AfxSig_end = 0,     // [marks end of message map]
#0004        AfxSig_vv,
#0005 };
#0006
#0007 #define ON_COMMAND(id, memberFxn) \
#0008   { WM_COMMAND, 0, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },
```

MFC.CPP

```
#0001  #include "my.h"//原该含入mfc.h就好，但为了extern CMyWinApp所以...
```

```
#0002
#0003 extern CMyWinApp theApp;
#0004 extern void printlpEntries(AFX_MSGMAP_ENTRY* lpEntry);
#0005
#0006 BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode)
#0007 {
#0008     // Now look through message map to see if it applies to us
#0009     AFX_MSGMAP* pMessageMap;
#0010     AFX_MSGMAP_ENTRY* lpEntry;
#0011     for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
#0012          pMessageMap = pMessageMap->pBaseMessageMap)
#0013     {
#0014          lpEntry = pMessageMap->lpEntries;
#0015          printlpEntries(lpEntry);
#0016     }
#0017
#0018     return FALSE;   // not handled
#0019 }
```

```
#0020
#0021  BOOL CWnd::Create()
#0022  {
#0023      cout << "CWnd::Create \n";
#0024      return TRUE;
#0025  }
#0026
#0027  BOOL CWnd::CreateEx()
#0028  {
#0029      cout << "CWnd::CreateEx \n";
#0030      PreCreateWindow();
#0031      return TRUE;
#0032  }
#0033
#0034  BOOL CWnd::PreCreateWindow()
#0035  {
#0036    cout << "CWnd::PreCreateWindow \n";
#0037    return TRUE;
#0038  }
#0039
#0040  LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
#0041  {
#0042      AFX_MSGMAP* pMessageMap;
#0043      AFX_MSGMAP_ENTRY* lpEntry;
#0044
#0045      if (nMsg == WM_COMMAND) // special case for commands
#0046      {
#0047          if (OnCommand(wParam, lParam))
#0048              return 1L; // command handled
#0049          else
#0050              return (LRESULT)DefWindowProc(nMsg, wParam, lParam);
#0051      }
#0052
#0053      pMessageMap = GetMessageMap();
#0054
#0055      for (; pMessageMap != NULL;
#0056           pMessageMap = pMessageMap->pBaseMessageMap)
#0057      {
#0058          lpEntry = pMessageMap->lpEntries;
#0059          printlpEntries(lpEntry);
#0060      }
#0061      return 0;  // add by JJHou. if find, should call lpEntry->pfn,
#0062             //           otherwise should call DefWindowProc.
#0063  }
#0064
#0065  LRESULT CWnd::DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam)
```

```
#0066 {
#0067     return TRUE;
#0068 }
#0069
#0070 BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
#0071 {
#0072     // ...
#0073     return OnCmdMsg(0, 0);
#0074 }
#0075
#0076 BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
#0077 {
#0078     // ...
#0079     // route as normal command
#0080     return CWnd::OnCommand(wParam, lParam);
#0081 }
```

```
#0082
#0083 BOOL CFrameWnd::Create()
#0084 {
#0085     cout << "CFrameWnd::Create \n";
#0086     CreateEx();
#0087     return TRUE;
#0088 }
#0089
#0090 BOOL CFrameWnd::PreCreateWindow()
#0091 {
```

```
#0092     cout << "CFrameWnd::PreCreateWindow \n";
#0093     return TRUE;
#0094 }
#0095
#0096 CView* CFrameWnd::GetActiveView() const
#0097 {
#0098     return m_pViewActive;
#0099 }
#0100
#0101 BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode)
#0102 {
#0103     // pump through current view FIRST
#0104     CView* pView = GetActiveView();
#0105     if (pView->OnCmdMsg(nID, nCode))
#0106         return TRUE;
#0107
#0108     // then pump through frame
#0109     if (CWnd::OnCmdMsg(nID, nCode))
#0110         return TRUE;
#0111
```

```
#0112     // last but not least, pump through app
#0113     CWinApp* pApp = AfxGetApp();
#0114     if (pApp->OnCmdMsg(nID, nCode))
#0115            return TRUE;
#0116
#0117     return FALSE;
#0118  }
#0119
#0120  BOOL CDocument::OnCmdMsg(UINT nID, int nCode)
#0121  {
#0122     if (CCmdTarget::OnCmdMsg(nID, nCode))
#0123         return TRUE;
#0124
#0125     return FALSE;
#0126  }
#0127
#0128  BOOL CView::OnCmdMsg(UINT nID, int nCode)
#0129  {
#0130     if (CWnd::OnCmdMsg(nID, nCode))
#0131         return TRUE;
#0132
#0133     BOOL bHandled = FALSE;
#0134     bHandled = m_pDocument->OnCmdMsg(nID, nCode);
#0135     return bHandled;
#0136  }
```

```
#0137
#0138  AFX_MSGMAP* CCmdTarget::GetMessageMap() const
#0139  {
#0140     return &CCmdTarget::messageMap;
#0141  }
#0142
#0143  AFX_MSGMAP CCmdTarget::messageMap =
#0144  {
#0145     NULL,
#0146     &CCmdTarget::_messageEntries[0]
#0147  };
```

```
#0148
#0149  AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
#0150  {
#0151
#0152          { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
#0153  };
#0154
#0155  BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
#0156  ON_COMMAND(CWndid, 0)
#0157  END_MESSAGE_MAP()
```

```
#0158
#0159  BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
#0160  ON_COMMAND(CFrameWndid, 0)
#0161  END_MESSAGE_MAP()
#0162
#0163  BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
#0164  ON_COMMAND(CDocumentid, 0)
#0165  END_MESSAGE_MAP()
#0166
#0167  BEGIN_MESSAGE_MAP(CView, CWnd)
#0168  ON_COMMAND(CViewid, 0)
#0169  END_MESSAGE_MAP()
#0170
#0171  BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
#0172  ON_COMMAND(CWinAppid, 0)
#0173  END_MESSAGE_MAP()
#0174
#0175  CWinApp* AfxGetApp()
#0176  {
#0177    return theApp.m_pCurrentWinApp;
#0178  }
#0179
```

```
#0180  LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
#0181                     CWnd *pWnd)  // last parameter pWnd is added by JJHou.
#0182  {
#0183    //...
#0184    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
#0185  }
#0186
#0187  LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
                              WPARAM wParam, LPARAM lParam)
#0188  {
#0189    LRESULT lResult = pWnd->WindowProc(nMsg, wParam, lParam);
#0190    return lResult;
#0191  }
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()   {
#0008                             }
```

```
#0009   CMyWinApp::~CMyWinApp()  {
#0010                           }
#0011   virtual BOOL InitInstance();
#0012   DECLARE_MESSAGE_MAP()
#0013 };
#0014
#0015 class CMyFrameWnd : public CFrameWnd
#0016 {
#0017 public:
#0018   CMyFrameWnd();
#0019   ~CMyFrameWnd()  {
#0020                  }
#0021   DECLARE_MESSAGE_MAP()
#0022 };
#0023
#0024 class CMyDoc : public CDocument
#0025 {
#0026 public:
#0027   CMyDoc::CMyDoc()  {
#0028                     }
#0029   CMyDoc::~CMyDoc() {
#0030                     }
#0031   DECLARE_MESSAGE_MAP()
#0032 };
#0033
#0034 class CMyView : public CView
#0035 {
#0036 public:
#0037   CMyView::CMyView()   {
#0038                        }
#0039   CMyView::~CMyView()  {
#0040                        }
#0041   DECLARE_MESSAGE_MAP()
```

```
#0042 };
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp;  // global object
#0004
#0005 BOOL CMyWinApp::InitInstance()
#0006 {
#0007     cout << "CMyWinApp::InitInstance \n";
#0008     m_pMainWnd = new CMyFrameWnd;
#0009     return TRUE;
```

```
#0010                           }
#0011   virtual BOOL InitInstance();
#0012   DECLARE_MESSAGE_MAP()
#0013 };
#0014
#0015 class CMyFrameWnd : public CFrameWnd
#0016 {
#0017 public:
#0018   CMyFrameWnd();
#0019   ~CMyFrameWnd()  {

#0020
#0021 BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0022 ON_COMMAND(CMyFrameWndid, 0)
#0023 END_MESSAGE_MAP()
#0024
#0025 BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
#0026 ON_COMMAND(CMyDocid, 0)
#0027 END_MESSAGE_MAP()
#0028
#0029 BEGIN_MESSAGE_MAP(CMyView, CView)
#0030 ON_COMMAND(CMyViewid, 0)
#0031 END_MESSAGE_MAP()
#0032
#0033 void printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
#0034 {
#0035 struct {
#0036   int classid;
#0037   char* classname;
#0038 } classinfo[] = {
#0039                   CCmdTargetid , "CCmdTarget   ",
#0040                   CWinThreadid , "CWinThread   ",

#0041                   CWinAppid    , "CWinApp      ",
#0042                   CMyWinAppid  , "CMyWinApp    ",
#0043                   CWndid       , "CWnd         ",
#0044                   CFrameWndid  , "CFrameWnd    ",
#0045                   CMyFrameWndid, "CMyFrameWnd  ",
#0046                   CViewid      , "CView        ",
#0047                   CMyViewid    , "CMyView      ",
#0048                   CDocumentid  , "CDocument    ",
#0049                   CMyDocid     , "CMyDoc       ",
#0050                   0            , "             "
#0051               };
#0052
#0053     for (int i=0; classinfo[i].classid != 0; i++)
#0054     {
#0055         if (classinfo[i].classid == lpEntry->nID)
```

```
#0056         {
#0057             cout << lpEntry->nID << "    ";
#0058             cout << classinfo[i].classname << endl;
#0059             break;
#0060         }
#0061     }
#0062 }
#0063 //------------------------------------------------------------------
#0064 // main
#0065 //------------------------------------------------------------------
#0066 void main()
#0067 {
#0068     CWinApp* pApp = AfxGetApp();
#0069
#0070     pApp->InitApplication();
#0071     pApp->InitInstance();
#0072     pApp->Run();
#0073
#0074     CMyDoc* pMyDoc = new CMyDoc;
#0075     CMyView* pMyView = new CMyView;
#0076     CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
#0077     pMyFrame->m_pViewActive = pMyView;
#0078     pMyView->m_pDocument = pMyDoc;
#0079
#0080     // test Message Routing
#0081     cout << endl << "pMyFrame received a WM_CREATE, routing path :" << endl;
#0082     AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
#0083
#0084     cout << endl << "pMyView received a WM_PAINT, routing path :" << endl;
#0085     AfxWndProc(0, WM_PAINT, 0, 0, pMyView);
#0086
#0087     cout << endl << "pMyView received a WM_COMMAND, routing path :" << endl;
#0088     AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
#0089
#0090     cout << endl << "pMyFrame received a WM_COMMAND, routing path :" << endl;
#0091     AfxWndProc(0, WM_COMMAND, 0, 0, pMyFrame);
#0092 }
```

本章回顾

像外科手术一样精准，我们拿起锋利的刀子，划开 MFC 坚韧的皮肤，再一刀下去，剖开它的肌理。掏出它的内脏，反复观察研究。终于，借着从 MFC 掏挖出来的原始代码清洗整理后完成的几个小小的C++console 程序，我们彻底了解了所谓Runtime Class、Runtime Time Information、Dynamic Creation、Message Mapping、Command Routing 的内部机制。

咱们并不是要学着做一套application framework，但是这样的学习过程确实有必要。因为，「只用一样东西，不明白它的道理，实在不高明」。况且，有什么比光靠三五个一两百行小程序，就搞定面向对象领域中的高明技术，更值得的事？有什么比欣赏那些由Runtime Class所构成的「类型录网」示意图、消息的实际流动图、消息映射表的架构图，更令人心旷神怡？

把Frame1~Frame8 好好研究一遍，你已经对MFC的架构成竹在胸。再来，就是MFC类的实际运用，以及Visual C++工具的熟练啰。

第三篇 浅出 MFC 程序设计

第 5 章 总观 Application Framework

> 带艺术气息的软件创作行为将在Application Framework 出现后逐渐成为工匠技术，而我们都将成为软件IC装配厂里的男工女工。
>
> 但，不是亨利福特，我们又如何能够享受大众化的汽车？
>
> 或许以后会出现「纯手工精制」的软件，可我自己从来不嫌机器馒头难吃。

什么是 Application Framework？

还没有学习任何一套Application Framework的使用之前，就给你近乎学术性的定义，我可以想象对你而言绝对是「形而上的」（超物质的无形哲理），尤其如果你对面向对象（Object Oriented）也还没有深刻体会的话。形而上者谓之道，形而下者谓之器，我想能够舍器而直接近道者，几稀！但是，「定义」这种东西又似乎宜开宗明义摆在前头。我诚挚地希望你在阅读后续的技术章节时能够时而回来看看这些形而上的叙述。当你有所感受，技术面应该也进入某个层次了。

侯捷怎么说

首先我们看看侯捷在其无责任书评中是怎么说的：

演化（evolution）永远在进行，但这个世界却不是每天都有革命性（revolution) 的事物发生。动不动宣称自己（或自己的产品）是划时代的革命性的，带来的影响就像时下满街跑的大 师一样使 我们渐渐 无动于衷（大师不可能满街跑）！但是Application Framework 的的确确在我们软件界称得上具有革命精神。

什么是Application Framework？Framework这个字眼有组织、框架、体制的意思，Application Framework不仅是一般性的泛称，它其实还是面向对象领域中的一个专有名词。

基本上你可以说，Application Framework是一个完整的程序模型，具备标准应用软件所需的一切基本功能，像是文件存取、打印预览、数据交换...，以及这些功能的使用接口（工具栏、状态栏、菜单、对话框）。如果更以术语来说，Application Framework 就是由一整组合作无间的「对象」架构起来的大模型。喔不不，当它还没有与你的程序产生火花的时候，它还只是有形无体，应该说是一组合作无间的「类」架构起来的大模型。

这带来什么好处呢？程序员只要带个购物袋到「类超级市场」采买，随你要买MDI或OLE或ODBC或Printing Preview，回家后就可以轻易拼凑出一个色香味俱全的大餐。

「类超级市场」就是C++ 类库，以产品而言，在Microsoft 是MFC，在Borland 是OWL，在IBM则是OpenClass。这个类库不只是类库而已，传统的函数库（C Runtime 或 Windows API）乃至于一般类库提供的是生鲜超市中的一条鱼一支葱一颗大白菜，彼此之间没有什么关联，主掌中馈的你必须自己选材自己调理。能够称得上Application Framework 者，提供的是火锅拼盘（就是那种带回家通通丢下锅就好的那种），依你要的是白菜火锅鱼头火锅或是麻辣火锅，菜色带调理包都给你配好。当然这样的火锅拼盘是不能够就地吃的，你得给它加点能量。放把火烧它吧，这火就是所谓的application object（在 MFC 程序中就是派生自CWinApp 的一个全局对象）。是这个对象引起了连锁反应（一连串的 'new'），使每一个形（类）有了真正的体（对象)，把应用程序以及 Application Framework 整个带动起来。一切因缘全由是起。

Application Framework 带来的革命精神是，程序模型已经存在，程序员只要依个人需求加料就好：在派生类中改写虚函数，或在派生类中加上新的成员函数。这很像你在火锅拼盘中依个人口味加盐添醋。

由于程序代码的初期规模十分一致（什么样风格的程序应该使用什么类，是一成不变的），而修改程序以符合私人需要的基本动作也很一致（我是指像「开辟一个空的骨干函数」这种事情)，你动不了 Application Framework 的大架构，也不需要动。这是福利不是约束。

应用程序代码骨干一致化的结果，使优越的软件开发工具如 CASE（Computer Aid Software Engineering）tool 容易开发出来。你的程序代码大架构掌握在 Application Framework 设计者手上，于是他们就有能力制作出整合开发环境（ Integrated Development Environment, IDE）了。这也是为什么 Microsoft、Borland、Symantec、Watcom、IBM 等公司的集成开发环境进步得如此令人咋舌的原因了。

有人说工学院中唯一保有人文气息的只剩建筑系，我总觉得信息系也勉强可以算上。带艺术气息的软件创作行为（我一直是这么认为的）将在 Application Framework 出现后逐渐成为工匠技术，而我们都将只是软件 IC 装配厂里的男工女工。其实也没什么好顾影自怜，功成名就的冠冕从来也不曾落在程序员头上；我们可能像纽约街头的普普（POP）工作者，自认为艺术家，可别人怎么看呢？不得而知！话说回来，把开发软件这件事情从艺术降格到工技，对人类只有好处没有坏处。不是亨利福特，我们又如何能够享受大众化的汽车？或许以后会出现「纯手工精制」的软件，谁感兴趣不得而知，我自己嘛...唔...倒是从来不嫌机器馒头难吃。

如果要三言两语点出 Application Framework 的特质，我会这么说：我们挖出别人早写好的一整套模块（MFC 或 OWL 或 OpenClass）之中的一部分，给个引子（application object）使它们一一具象化动起来，并被允许修改其中某些零件使这程序更符合私人需求，如是而已。

我怎么说

侯捷的这一段话实在已经点出Application Framework的精神。凝聚性强、组织化强的类库就是 Application Framework。一组合作无间的对象，彼此藉消息的流动而沟通，并且互相调用对方的函数以求完成任务，这就是 Application Framework。对象存在哪里？

在MFC中?! 这样的说法不是十分完善，因为MFC的各个类只是「对象属性（行为）的定义」而已，我们不能够说 MFC 中有实际的对象存在。唯有当程序被application object（这是一个派生自 MFC CWinApp 的全局对象）引爆了，才将我们选用的类一一具象化起来，产生实体并开始动作。图 5-1 是一个说明。

这样子说吧，静态情况下MFC 是一组类库，但在程序运行时期它就生出了一群有活动力的对象组。最重要的一点是，这些对象之间的关系早已建立好，不必我们（程序员）操心。好比说当用户按下菜单的【File/Open】项，开文件对话框就会打开；用户选好文件名后，Application Framework 就开始对着你的数据类，唤起一个名为Serialize 的特殊函数。这整个机制都埋好了，你只要把心力放在那个叫作 Serialize 的函数上即可。

选用标准的类，做出来的产品当然就没有什么特色，因为别人的零件和你的相同，兜起来的成品也就一样。我指的是用户接口（UI）对象。但你要知道，软件工业发展到现阶段这个世代，着重的已不再是UI 的争奇斗艳，取巧哗众；UI 已经渐渐走上标准化了。软件一决胜负的关键在数据的处理。事实上，在「真正做事」这一点，整个application framework 是无能为力的，也就是说对于数据结构的安排，数据的处理，数据的显示，Application Framework 所能提供的，无一不是单单一个空壳而已——在 C++ 语言来讲就是个虚函数。软件开发人员必须想办法改造（override）这些虚函数，才能符合个人所需。基于 C++ 语言的特性，我们很容易继承既有之类并加上自己的特色，这就是面向对象程序设计的主要精神。也因此，C++ 语言中有关于「继承」性质的份量，在MFC程序设计里头占有很重的比例，在学习使用MFC 的同时，你应该对C++的继承性质和虚函数有相当的认识。第 2 章有我个人对C++这两个性质的心得。

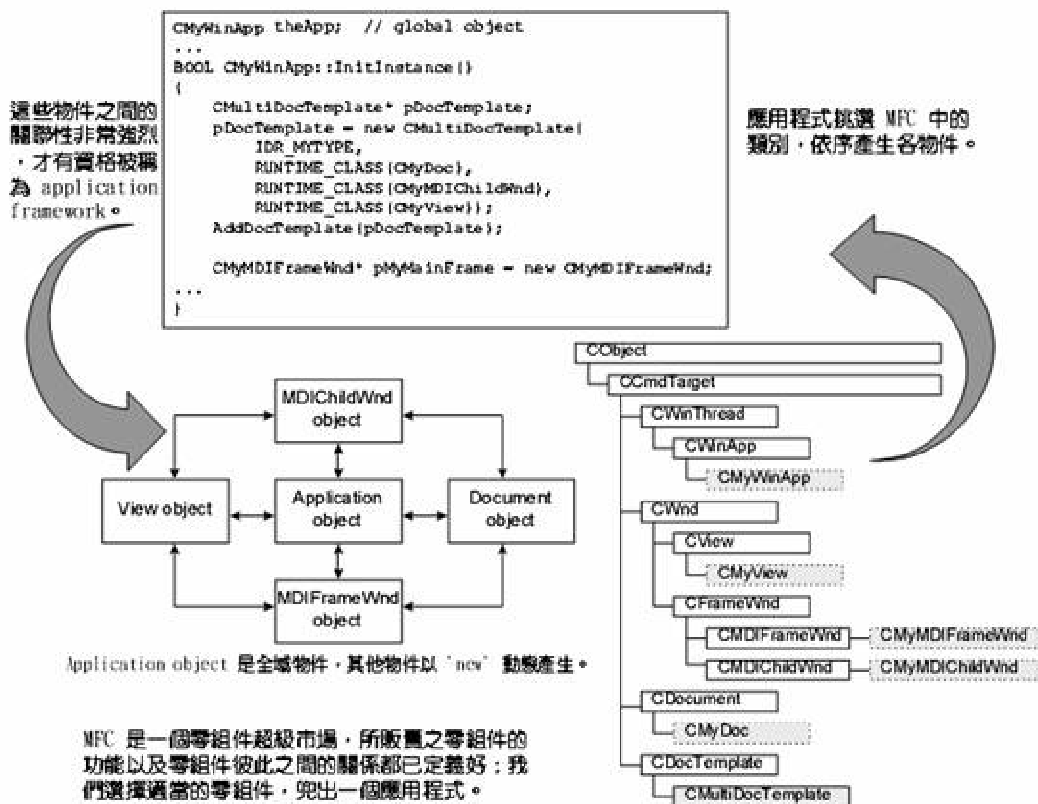


图5-1 MFC 是一个零组件超级市场，所贩卖的零组件功能以及零组件

彼此之间的关系都已定义好；我们选择自己喜欢的零件，兜出一个应用程序

Application Framework 究竟能提供我们多么实用的类呢？或者我们这么问：哪些工作已经被处理掉了，哪些工作必须由程序员扛下来？比方说一个标准的 MFC 程序应该有一个可以读写文件数据的功能，然而应用程序本身有它独特的数据结构，从 MFC 得来的零件可能与我的个人需求搭配吗？

我以另一个比喻做回答。

假设你买了一个整厂整线计划，包括仓贮、物料、MIS、生管，各个部门之间的搭配也都建立起来了，包含在整厂整线计划内。这个厂原先是为了生产葡萄酒，现在改变主意要生产白兰地，难道整个厂都不能用了吗？不！只要把进货原料改一改、发酵程序改一改、瓶装程序改一改、整个厂的其它设备以及设备与设备之间的联机配合（这是顶顶重要的）都可以再利用。物料之后到生管，装瓶之后到仓贮，仓贮之后到出货，再加上 MIS监控全厂，这些程序都是不必改变的。

一个整厂整线计划，每一单元之间的联机沟通，合作关系，都已经建立起来，是一种构造好的运作模式。抽换某个单元的性质或部分性质，并不影响整体作业。「整厂整线」最重要最有价值的是各单元之间的流程与控制。

反映到面向对象程序设计里头，Application Framework 就是「整厂整线规划」，Application Framework 提供的类就是上述工厂中一个一个的单元。最具价值的就是各类之间的交互运作模式。

虽然文件读写（此动作在MFC 称为Serialize）单元必须改写以符合个人所需，但是单元与单元之间的关系依然存在而且极富价值。当用户在程序中选按【File/Open】或【File/Save】，框架窗口自动通知Document 对象（内存数据），引发Serialization 动作；而你为了个人的需求，改写了这个 Serialize 虚函数。你对 MFC 的改写范围与程度，视你的程序有多么特异而定。

可是如果酿酒厂想改装为炼钢厂？那可就无能为力了。这种情况不会出现在软件开发上，因为软件的必备功能使它们具有相当的相似性（尤其 Windows 又强调界面一致性)。好比说程序想支持 MDI 接口，想支持 OLE，这基本上是超越程序之专业应用领域之外的一种大格局，一种大架构，最适合 Application Framework 发挥。

很明显，Application Framework是一组超级的类库。能够被称为 Framework 者必须其中的类性质紧密咬合，互相呼应。因此你也就可以想象，Framework 所提供的类是一伙的，不是单片包装的。你把这伙东西放入程序里，你也就得乖乖遵循一种特定的（Application Framework 所规定的）程序风格来进进程序设计工作。但是侯捷也告诉我们，这是福利不是约束。

别人怎么说

其它人又怎么看 Application Framework？我将胪列数篇文章中的相关定义。若将原文译为中文，我恐怕力有未逮辞不达意，所以列出原文供你参考。

1985 年，Apple公司的MacApp严格而系统化地定义出做为一个商业化 Application Framework 所需要的关键理念：

> The key ideas of a commercial application framework : a generic app on steroids that provides a large amount of general-purpose functionality within a well-planned, well-tested, cohesive structure.

cohesive 的意思是强而有力、有凝聚力的。

steroid 是类固醇。自从加拿大 100 公尺名将班琼森在汉城奥运吃了这药物而夺得金牌并打破世界记录，相信世人对这个名称不会陌生（当然琼森的这块金牌和他的世界记录后来是被取消的）。

类固醇俗称美国仙丹，是一种以胆固醇结构为基础，派生而来的荷尔蒙，对于发炎红肿等症状有极速疗效。然而因为它是透过抑制人类免疫系统而得到疗效，如果使用不当，会带来极不良的副作用。

运动员用于短时间内增强身体机能的雄性激素就是类固醇的一种，会影响脂肪代谢，服用过量会导至极大的副作用。

基本上MacApp以类固醇来比拟Application Framework虽是妙喻，但类固醇会对人体产生不好的副作用而 Application Framework 不会对软件开发产生副作用-- 除非你认为不能随心所欲写你的代码也算是一种副作用。

Apple 更一步更明确地定义一个Application Framework 是：

> an extended collection of classes that cooperate to support a complete application architecture or application model, providing more complete application development support than a simple set of class libraries.

这里所指的 support 并不只是视觉性 UI 组件如menu、dialog、listbox...，还包括一个应用程序所需要的其它功能设备，像是 Document, View, Printing, Debugging。

另一个相关定义出现在 Ray Valdes 于1992年10月发表于Dr. Dobb's Journal 的 "Sizing up Application Frameworks and Class Libraries" 一文之中：

An application framework is an integrated object-oriented software system that offers all the application-level classes (documents, views, and commands) needed by a generic application.

An application framework is meant to be used in its entirety, and fosters both design reuse and code reuse. An application framework embodies a particular philosophy for structuring an application, and in return for a large mass of prebuilt functionality, the programmer gives up control over many architectural-design decisions.

Donald G. Firesmith 在一篇名为 "Frameworks : The Golden path of the object Nirvana" 的文章中对 Application Framework 有如下定义：

What are frameworks? They are significant collections of collaborating classes that capture both the small-scale patterns and major mechanisms that, in turn, implement the common requirements and design in a specific application domain.

* Nirvana 是涅盘、最高境界的意思。

Bjarne Stroustrup（C++ 原创者）在他的 The C++ Programming Language 一书中对于 Application Framework 也有如下叙述：

Libraries build out of the kinds of classes described above support design and re-use of code by supplying building blocks and ways of combining them; the application builder designs a framework into which these common building blocks are fitted. An alternative, and sometimes more ambitious, approach to the support of design and re-use is to provide code that establishes a common framework into which the application builder fits application-specific code as building blocks. Such an approach is often called an application framework. The classes establishing such a framework often have such fat interfaces that they are hardly types in the traditional sense. They approximate the ideal of being complete applications, except that they don't do anything. The specific actions are supplied by the application programmer.

Kaare Christian 在1994/02/08的PC Magazine 中有一篇"C++ Application Frameworks" 文章，其中有下列叙述（节录）：

两年前我在纽约北边的乡村盖了一栋 post-and-beam 房子。在我到达之前我的木匠已经把每一根梁的外形设计好并制作好，把一根根的粗糙木材变成一块块锯得漂漂亮亮的零件，一切准备就绪只待安装。（注：所谓post-and-beam 应是指那种梁柱都已规格化，可以邮购回来自己动手盖的DIY——Do It Yourself——房子）。

使用Application Framework建造一个Windows 应用程序也有类似的过程。你使用一组早已做好的零件，它使你行进快速。由于这些零件坚强耐用而且稳固，后面的工作就简单多了。但最重要的是，不论你使用规格化的梁柱框架来盖一栋房子，或是使用Application Framework来建立一个 Windows 程序，工作类型已然改变，出现了一种完全崭新的做事方法。在我的post-and-beam 房子中，工作类型的改变并不总是带来帮助；贸易商在预制梁柱的技巧上可能会遭遇适应上的困扰。同样的事情最初也发生在Windows 身上，因为你原已具备的某些以C 语言写 Windows 程序的能力，现在在以C++ 和Application Framework 开发程序的过程中无用武之地。时间过去之后，Windows 程序设计的类型移转终于带来了伟大的利益与方便。Application Framework本身把message loops 和其它Windows的苦役都做掉了，它促进一个比较秩序井然的程序结构。

Application Framework——建立Windows应用软件所用的C++类库——如今已行之有年，因为面向对象程序设计已经快速地获得了接受度。Windows API 是程序性的，Application Framework 则让你写面向对象式的Windows程序。它们提供预先写好的机能（以 C++ 类型式呈现出来），可以加速应用软件的开发。

Application Framework 提供数种优点。或许最重要的，是它们在面向对象程序设计模式下对Windows 程序设计过程的影响。你可以使用 Framework 来减轻例行但繁复的琐事，目前的Application Framework 可以在图形、对话框、打印、求助、OCX 控制组件、剪贴板、OLE等各方面帮助我们，它也可以产生漂亮的UI接口如工具栏和状态栏。

借着Application Framework的帮助写出来的代码往往比较容易组织化，因为 Framework改变了Windows管理消息的方法。也许有一天Framework还可以帮你维护单一一套代码以应付不同的执行平台。

你必须对Application Framework有很好的知识，才能够修改由它附带的软件开发工具制作出来的骨干程序。它们并不像Visual Basic那么容易使用。但是对Application Framework 专家而言，这些程序代码产生器可以省下大量时间。

使用Application Framework的主要缺点是，没有单一一套产品广被所有的C++编译器支持。所以当你选定一套 Framework，在某个范围来说，你也等于是选择了一个编译器。

为什么使用 Application Framework

虽然Application Framework 并不是新观念，它们却在最近数年才成为 PC 平台上软件开发的主流工具。面向对象语言是具体实现Application Framework 的理想载具，而C++ 编译器在PC平台上的出现与普及终于允许主流PC程序员能够享受Application Framework 带来的利益。

从八十年代早期到九十年代初始，C++大都存在于UNIX 系统和研究人员的工作站中，不在PC 以及商业产品上。C++ 以及其它的面向对象语言（例如 Smalltalk-80）使一些大学和研究计划生产出现今商业化Application Framework 的鼻祖。但是这些早期产品并没有明显区隔出应用程序与 Application Framework 之间的界线。

今天应用软件的功能愈来愈复杂，建造它们的工具亦复如此。Application Framework、Class Library 和GUI toolkits 是三大类型的软件开发工具（请见方块说明），这三类工具虽然以不同的技术方式逼近目标，它们却一致追求相同而基本的软件开发关键利益：降低写程序代码所花的精力、加速开发效率、加强可维护性、增加强固性（robustness）、为组合式的软件机能提供杠杆支点（有了这个支点，再大的软件我也举得起来）。

当我们面临软件工业革命，我们的第一个考虑点是：我的软件开发技术要从哪一个技术面切入？从raw API 还是从高阶一点的工具？如果答案是后者，第二个考虑点是我使用哪一层级的工具？GUI toolkits 还是Class Library 还是 Application Framework？如果答案又是后者，第三个考虑点是我使用哪一套产品？MFC 或 OWL 或 Open Class Library？

（目前 PC 上还没有第四套随编译器附赠的 Application Framework 产品）

别认为这是领导者的事情不是我（工程师）的事情，有这种想法你就永远当不成领导者。也别认为这是工程师的事情不是我（学生）的事情，学生的下一步就是工程师；及早想点工业界的激烈竞争，对你在学生阶段规划人生将有莫大帮助。

我相信，Application Framework 是最好的杠杆支点。

Application Framework，Class Library，GUI toolkit

一般而言，Class Library 和 GUI toolkit 比 Application Framework 的规模小，定位也没那么高阶宏观。Class Library 可以定义为「一组具备面向对象性质的类，它们使应用程序的某些功能实现起来容易一些，这些功能包括数值运算与数据结构、绘图、内存管理等；这些类可以一片一片毫无瓜葛地并入应用程序内」。

请特别注意这个定义中所强调的「一片一片毫无瓜葛」，而不像 Application Framework 是大伙儿一并加入。因此，你尽可以随意使用 Class Library，它并不会强迫你遵循任何特定的程序架构。Class Library 通常提供的不只是 UI 功能、也包括一般性质的机能，像数据结构的处理、日期与时间的转换等等。

GUI toolkit 提供的服务类似Class Library，但它的程序接口是程序导向而非面向对象。而且它的功能大都集中在图形与 UI 接口上。GUI toolkit 的发展历史早在面向对象语言之前，某些极为成功的产品甚至是以汇编语言（assembly）写成。不要必然地把GUI 联想到Windows，GUI toolkit也有DOS 版本。我用过的Chatter Box 就是 DOS 环境下的 GUI 工具（是一个函数库）。

使用 Application Framework 的最直接原因是，我们受够了日益暴增的 Windows API。把 MFC 想象为第四代语言，单单一个类就帮我们做掉原先要以一大堆 APIs 才能完成的事情。

但更深入地想，Application Framework绝不只是为了降低我们花在浩瀚无涯的Windows API的时间而已；它所带来的面向对象程序设计观念与方法，使我们能够站在一群优秀工程师（MFC 或 OWL 的创造者）的努力心血上，继承其成果而开发自己之所需。同时，因为Application Framework 特殊的工作类型，整体开发工具更容易制作，也制作的更完美。在我们决定使用 Application Framework 的同时，我们也获得了这些整合性软件开发环境的支持。在软件开发过程中，这些开发工具角色之重要性不亚于 Application Framework 本身。

Application Framework 将成为软件技术中最重要的一环。如果你不知道它是什么，赶快学习它；如果你还没有使用它，赶快开始用。机会之窗不会永远为你打开，在你的竞争者把它关闭之前赶快进入！如果你认为改朝换代还早得很，请注意两件事情。第一，江山什么时候变色可谁也料不准，当你埋首工作时，外面的世界进步尤其飞快；第二，面向对象和Application Framework可不是那么容易学的，花多少时间才能登堂入室可还得凭各人资质和基础呢。

浩瀚无涯的 Windows API

Windows 版本	推出日期	API 個數	訊息個數
1.0	1985.11	379	?
2.0	1987.11	458	?
3.0	1990.05	578	?
Multimedia Ex.	1991.12	120	?
3.1	1992.04	973	271
Win32s	1993.08	838	287
Win32	1993.08	1449（持續增加當中）	291（持續增加當中）

Microsoft Foundation Classes （MFC）

PC 世界里出了三套C++ Application Frameworks，并且有越来越多的趋势。这三套是Microsoft的MFC （ Microsoft Foundation Classes ） ， Borland 的 OWL（ Object WindowLibrary），以及IBM VisualAge C++ 的Open Class Library。至于其它 C++ 编译器厂商如 Watcom、Symantec、Metaware，只是供应集成开发环境（Integraded Development Environment，IDE），其 Application Framework 都是采用微软公司的 MFC。

Delphi（Pascal 语言），依我之见，也称得上是一套 Application Framework。Java 语言本身内建一套标准类库，依我之见，也够得上资格被称为 Application Framework。

Delphi 和Visual Basic，又被称为是一种应用程序快速开发工具（RAD，Rapid Application Development）。它们采用PME（Properties-Method-Event）架构，写程序的过程像是在一张画布上拼凑一个个现成的组件（components）：设定它们的属性（properties）、指定它们应该「有所感」的外来刺激（events），并决定它们面对此刺激时在预设行为之外的行为（methods）。所有动作都以拖拉、设定数值的方式完成，非常简单。只有在设定组件与组件之间的互动关系时才牵涉到程序代码的写作（这一小段代码也因此成为顺利成功的关键）。

Borland 公司于1997 年三月推出的C++ Builder也属于PME架构，提供一套Visual Component Library（VCL），内有许许多多的组件。因此 C++ Builder 也算得上是一套RAD（应用程序快速开发工具）。

早初，开发Windows应用程序必须使用微软的SDK（Software Development Kit），直接调用Windows API 函数，向Windows 操作系统提出各种要求，例如配置内存、开启窗口、输出图形...。

所谓API（Application Programming Interface），就是开放给应用程序调用的系统功能。

数以千计的Windows APIs，每个看起来都好像比重相若（至少你从手册上看不出来孰轻孰重）。有些APIs 彼此虽有群组关系，却没有相近或组织化的函数名称。星罗棋布，雾列星驰；又似雪球一般愈滚愈多，愈滚愈大。撰写 Windows 应用程序需要大量的耐力与毅力，以及大量的小心谨慎！

MFC帮助我们把这些浩繁的APIs，利用面向对象的原理，逻辑地组织起来，使它们具备抽象化、封装化、继承性、多态性、模块化的性质。

1989 年微软公司成立Application Framework 技术团队，名为AFX小组，用以开发C++ 面向对象工具给 Windows 应用程序开发人员使用。AFX 的 "X" 其实没有什么意义，只是为了凑成一个响亮好念的名字。

这个小组最初的「宪章」，根据记载，是要 "utilize the latest in object oriented technology to provide tools and libraries for developers writing the most advanced GUI applications on the market"，其中并未画地自限与Windows 操作系统有关。果然，其第一个原型产品，有自己的窗口系统、自己的绘图系统、自己的对象数据库、乃至于自己的内存管理系统。

当小组成员以此产品开发应用程序，他们发现实在是太复杂，又悖离公司的主流系统——Windows——太遥远。于是他们修改宪章变成 "deliver the power of object-oriented solutions to programmers to enable them to build world-class Windows based applications in C++." 这差不多正是Windows 3.0 异军崛起的时候。

C++ 是一个复杂的语言，AFX小组预期MFC的用户不可能人人皆为C++ 专家，所以他们并没有采用所有的C++ 高阶性质（例如多重继承）。许多「麻烦」但「几乎一成不变」的Windows程序动作都被隐藏在MFC类之中，例如WinMain 、 RegisterClass、Window Procedure 等等等。

虽说这些被隐藏的Windows 程序动作几乎是一成不变的，但它们透露了 Windows 程序的原型奥秘，这也是为什么我要在本书之中锲而不舍地挖出它们的原因。

为了让MFC尽可能地小，尽可能地快，AFX 小组不得不舍弃高度的抽象（导至过多的虚函数），而引进他们自己发明的机制，尝试在面向对象领域中解决 Windows 消息的处理问题。这也就是本书第 9 章深入探讨的Message Mapping 和Message routing 机制。注意，他们并没有改变C++语言本身，也没有扩大语言的功能。他们只是设计了一些令人拍案叫绝的宏，而这些宏背后隐藏着巨大的机制。

了解这些宏（以及它们背后所代表的机制）的意义，以及隐藏在 MFC 类之中的那些足以曝露原型机密的「麻烦事儿」，正是我认为掌握 MFC 这套 Application Framework的重要手段。

就如同前面那些形而上的定义，MFC 是一组凝聚性强、组织性强的类库。如果你要利用MFC发展你的应用程序，必须同时引用数个必要类，互相搭配支持。图5-3是一个标准的MFC程序外貌。隐藏在精致画面背后更重要的是，就如我在前面说过，对象与对象之间的关系已经存在，消息的流动程序也都已设定。当你要为这个程序设计真正的应用功能，不必在意诸如「我如何得知用户按左键？左键按下后我如何启动某一个函数？参数如何传递过去...」等琐事，只要专注在左键之后真正要做的功能动作就好。

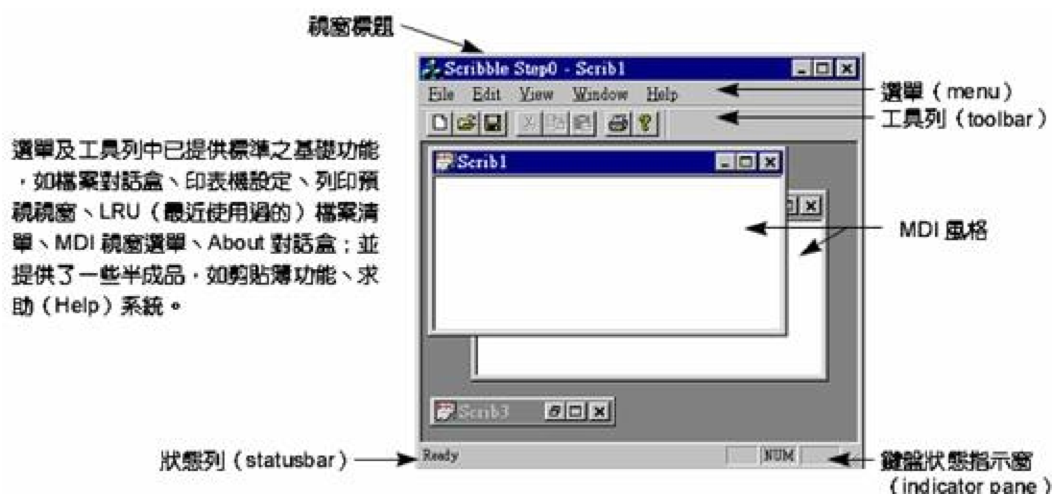


图 5-3 标准MFC程序的风貌

白头宫女话天宝：Visual C++ 与 MFC

微软公司于1992/04 推出C/C++ 7.0 产品时初次向世人介绍了MFC 1.0，这个初试啼声的产品包含了20,000行C++原始代码，60个以上的Windows相关类，以及其它的一般类如时间、数据处理、文件、内存、诊断、字符串等等等。它所提供的，其实是一个 "thin and efficient C++ transformation of the Windows API"。其32 位版亦在1992/07 随着 Win32 SDK 推出。

MFC 1.0 获得的回响带给 AFX 小组不少鼓舞。他们的下一个目标放在：

- 更高阶的架构支持
- 罐装组件（尤其在用户接口上）

前者成就了Document/View 架构，后者成就了工具栏、状态栏、打印、预览等极受欢迎的UI性质。当然，他们并没有忘记兼容性与移植性。虽然 AFX 小组并未承诺MFC可以跨不同操作系统如 UNIX XWindow、OS/2 PM、Mac System 7，但在其本家（Windows 产品线）身上，在16位Windows 3.x和32位Windows 95 与Windows NT之间的移植性是无庸置疑的。虽然其16位产品和32位产品是分别包装销售，你的原始代码通常只需重新编译链接即可。

Visual C++ 1.0（也就是 C/C++ 8.0）搭配 MFC 2.0 于1993/03 推出，这是针对Windows 3.x 的16位产品。接下来又在1993/08 推出在Windows NT上的 Visual C++ 1.1 for Windows NT，搭配的是MFC 2.1。这两个版本有着相同的基本性质。MFC 2.0 内含近60,000 行 C++ 程序代码，分散在 100 个以上的类中。Visual C++ 整合环境的数个重要工具（大家熟知的 Wizards）本身即以 MFC 2.0 设计完成，它们的出现对于软件生产效率的提升有极大贡献。

微软在 1993/12 又推出了 16 位的 Visual C++ 1.5，搭配 MFC 2.5。这个版本最大的进步是多了 OLE2 和 ODBC 两组类。整合环境也为了支持这两组类而做了些微改变。

1994/09，微软推出Visual C++ 2.0，搭配MFC 3.0，这个32位版本主要的特征在于配合目标操作系统（Windows NT 和Windows 95），支持多线程。所有类都是thread-safe。UI 对象方面，加入了属性表（Property Sheet）、miniframe 窗口、可随处停驻的工具栏。MFC collections 类改良为 template-based。链接器有重大突破，原使用的Segmented Executable Linker 改为 Incremental Linker，这种链接器在对 OBJ 文件做链接时，并不每次从头到尾重新来过，而只是把新数据往后加，旧数据加记作废。想当然耳，EXE 文件会累积许多不用的垃圾，那没关系, 透过Win32 memory-mapped file, 操作系统（Windows NT 及 Windows 95）只把欲使用的部分加载，丝毫不影响执行速度。必要时程序员也可选用传统方式链接，这些垃圾自然就不见了。对我们这些终日受制于edit-build-run-debug 轮回的程序员, Incremental Linker 可真是个好礼物。

1995/01，微软又加上了MAPI（Messaging API）和 WinSock 支持，推出 MFC 3.1（32 位元版），并供应13个通用控制组件，也就是 Windows 95 所提供的tree、tooltip、spin、slider、progress、RTF edit 等等控制组件。

1995/07，MFC 有了3.2 版，那是不值一提的小改版。

然后就是1995/09 的32 位MFC 4.0。这个版本纳入了DAO 数据库类、多线程同步控制类，并允许制作 OCX containers。搭配推出的 Visual C++ 4.0 编译器，也终于支持了 template、RTTI 等 C++ 语言特性。IDE 整合环境有重大的改头换面行动，Class View、Resource View、File View 都使得项目的管理更直觉更轻松，Wizardbar 则活脱脱是一个简化的 ClassWizard。此外，多了一个极好用的Components Gallery，并允许程序员订制 AppWizard。

1996 年上半年又推出了MFC 4.1，最大的焦点在ISAPI（Internet Server API）的支持，提供五个新类， 分别是CHttpServer、CHttpFilter、 CHttpServerContext、CHttpFilterContext、CHtmlStream，用以建立交互式 Web 应用程序。整合环境方面也对应地提供了一个ISAPI Extension Wizard。在附加价值上，Visual C++ 4.1提供了Game SDK，帮助开发Windows 95上的高效率游戏软件。Visual C++ 4.1 还提供不少个由协力公司完成的 OLE控制组件（OCXs），这些 OLE 控制组件技术很快就要全面由桌上跃到网上，称为 ActiveX 控制组件。不过，遗憾的是，Visual C++ 4.1 的编译器有些臭虫，不能够制作 VxD（虚拟装置驱动程序）。

1996 年下半年推出的 MFC 4.2，提供对 ActiveX 更多的技术支持，并整合 Standard C++ Library。它封包一组新的 Win32 Internet 类（统称为 WinInet），使 Internet 上的程序开发更容易。它提供 22 个新类和 40 个以上的新成员函数。它也提供一些控制元件，可以系结（binding）近端和远程的数据源（data sources）。整合环境方面，Visual C++ 4.2提供新的Wizard给ActiveX 程序开发使用，改善了影像编辑器，使它能够处理在Web 服务器上的两个标准图文件格式：GIF 和 JPEG。

1997 年五月推出的Visual C++ 5.0，主要诉求在编译器的速度改善，并将Visual C++ 合并到微软整个Visual Tools的终极管理软件Visual Studio 97 之中。所有的微软虚拟开发工具，包括Visual C++、Visual Basic、Visual J++、Visual InterDev、Visual FoxPro、都在Visual Studio 97 的整合之下有更密切的彼此奥援。至于程序设计方面，MFC 本身没有什么变化（4.21 版），但附了一个 ATL（Active Template Library）2.1 版，使 ActiveX 控制组件的开发更轻松些。

我想你会发现，微软正不断地为「为什么要使用MFC」加上各式各样的强烈理由，并强烈导引它成为Windows 程序设计的 C++ 标准接口。你会看到愈来愈多的 MFC/C++ 程序代码。对于绝大多数的技术人员而言，Application Framework 的抉择之道无它，「MFC是微软公司钦定产品」，这个理由就很呛人了。

纵览 MFC

MFC非常巨大（其它application framework也不差），在下一章正式使用它之前，让我们先做个浏览。

MFC 类主要可分为下列数大群组：

- General Purpose classes——提供字符串类、数据处理类（如数组与串列），异常情况处理类、文件类...等等。
- Windows API classes - 用来封包Windows API，例如窗口类、对话框类、DC 类...等等。
- Application framework classes——组成应用程序骨干者，即此组类，包括 Document/View、消息捕获、消息映射、消息循环、动态生成、文件读写等等。
- High level abstractions - 包括工具栏、状态栏、分裂窗口、卷动窗口等等。
- operation system extensions - 包括OLE、ODBC、DAO、MAPI、WinSock、ISAPI等等。

General Purpose classes

也许你使用 MFC 的第一个目标是为了写 Windows 程序，但并不是整个 MFC 都只为此目的而活。下面这些类适用于 Windows，也适用于 DOS。

CObject

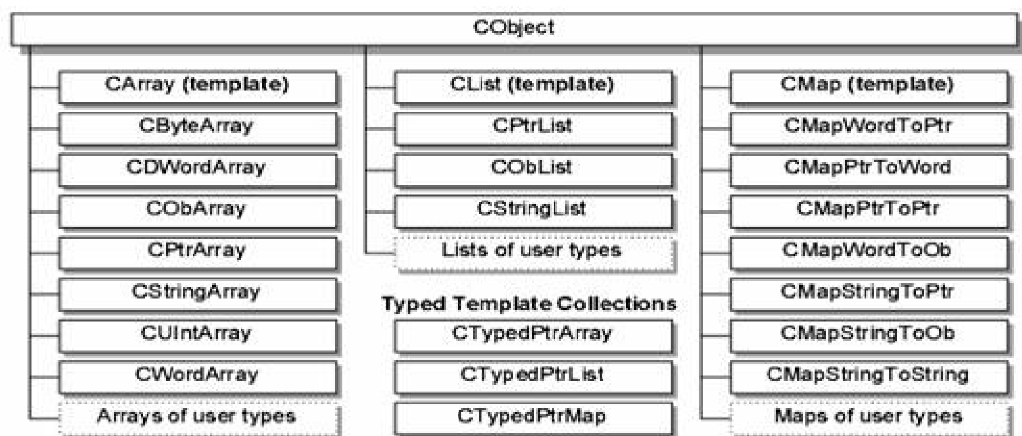
绝大部分类库，往往以一个或两个类，做为其它绝大部分类的基础。MFC 亦复如此。CObject是万类之首，凡类派生自CObject者，得以继承数个面向对象重要性质，包括RTTI（运行时期类型识别）、Persistence（对象保存）、Dynamic Creation（动态生成）、Diagnostic（错误诊断）。本书第 3 章对于这些技术已有了一份 DOS 环境下的模拟，第 8 章另有 MFC 相关原始代码的探讨。其中，「对象保存」又牵扯到 CArchive，「诊断」又牵扯到 CDumpContext，「运行时期类型识别」以及「动态生成」又牵扯到CRuntimeClass。

数据处理类（collection classes）

所谓collection，意指用来管理一「群」对象或标准类型的数据。这些类像是Array或List 或 Map 等等，都内含针对元素的「加入」或「删除」或「巡访」等成员函数。Array （数组）和 List（串列）是数据结构这门课程的重头戏，大家比较熟知，Map（可视之为表格）则是由成双成对的两两对象所构成，使你很容易由某一对象得知成对的另一对象；换句话说一个对象是另一个对象的键值（key）。例如，你可以使用 String-to-String Map，管理一个「电话-人名」数据库；或者使用 Word-to-Ptr Map，以 16 位数值做为一个指针的键值。

最令人侧目的是，由于这些类都支持 Serialization，一整个数组或串列或表格可以单一一进程序代码就写到文件中（或从文件读出）。第 8 章的 Scribble Step1范例程序中你就会看到它的便利。

MFC 支持的collection classes有：



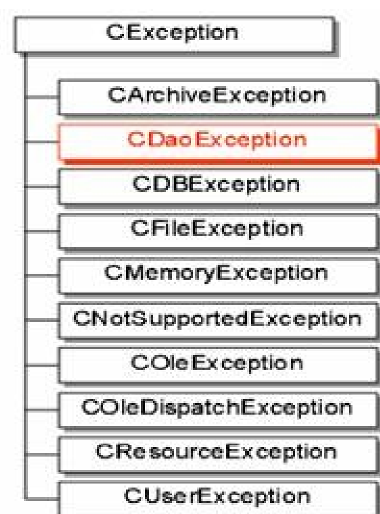
杂项类

- Crect——封装Windows 的RECT 结构。这个类在Windows环境中特别有用，因为CRect常常被用作MFC类成员函数的参数。
- Csize——封装Windows的SIZE结构。

- Cpoint——封装Windows的POINT 结构。这个类在Windows环境中特别有用，因为CPoint常常被用作MFC类成员函数的参数。
- Ctime——表现绝对时间，提供许多成员函数，包括取得目前时间 （static GetCurrentTime）、将时间数据格式化、抽取特定字段（时、分、秒）等等。它对于 +、-、+=、-+ 等运算符都做了重载动作。
- CtimeSpan——以秒数表现时间，通常用于计时秒表。提供许多成员函数，包括把秒数转换为日、时、分、秒等等。
- Cstring——用来处理字符串。支持标准的运算符如 =、+=、< 和 >。

异常处理类（exception handling classes）

所谓异常情况（exception），是发生在你的程序运行时期的不正常情况，像是文件打不开、内存不足、写入失败等等等。我曾经在第 2 章最后面介绍过异常处理的观念及相关的MFC类，并在第 4 章「Exception Handling」一节介绍过一个简单的例子。与「异常处理」有关的MFC类一共有以下11种：



Windows API classes

这是MFC声名最大的一群类。如果你去看看原始代码，就会看到这些类的成员函数所对应的各个Windows API函数。

- CwinThread——代表MFC程序中的一个线程。自从3.0 版之后，所有的MFC类就都已经是thread-safe了。SDK 程序中标准的消息循环已经被封装在此一类之中（你会在第 6 章看到我如何把这一部分开膛剖肚）。

- CwinApp——代表你的整个MFC应用程序。此类派生自CWinThread；要知道，任何32位Windows程序至少由一个线程构成。CWinApp 内含有用的成员变 数 如m_szExeName，放置执行文件文件名，以及有用的成员函数如ProcessShellCommand，处理命令列选项。
- CWnd——所有窗口，不论是框架窗口、子框窗口、对话框、控制组件、view 窗口，都有一个对应的C++ 类，你可以想象「窗口handle」和「C++ 对象」结盟。这些C++ 类统统派生自 CWnd，也就是说，凡派生自 CWnd 之类才能收到 WM_ 窗口消息（WM_COMMAND 除外）。

  所谓「窗口handle」和「C++ 对象」结盟，实际上是CWnd 对象有一个成员变量m_hWnd，就放着对应的窗口handle。所以，只要你手上有一个CWnd 对象或CWnd 对象指针，就可以轻易获得其窗口handle：

  ```
  HWND hWnd = pWnd->m_hWnd;
  ```

- CcmdTarget——CWnd的父类。派生自它， 类才能够处理命令消息WM_COMMAND。这个类是消息映射以及命令消息循环的大部分关键，我将在第 9 章推敲这两大神秘技术。
- GDI 类、DC 类、Menu 类。

Application framework classes

这一部分最为人认知的便是Document/View，这也是使MFC跻身 application framework的关键。Document/View 的观念是希望把数据的本体，和数据的显示分开处理。由于文件产生之际，必须动态生成 Document/View/Frame 三种对象，所以又必须有所谓的Document Template 管理之。

CDocTemplate、CSingleDocTemplate、CmultiDocTemplate——Document Template 扮演黏胶的角色，把 Document 和 View 和其 Frame（外框窗口）胶黏在一块儿。CSingleDocTemplate 一次只支持一种文件类型，CMultiDocTemplate 可同时支持多种文件类型。注意，这和 MDI 程序或 SDI 程序无关，换句话说，MDI 程序也可以使用CSingleDocTemplate，SDI 程序也可以使用CMultiDocTemplate。

但是，逐渐地，MDI 这个字眼与它原来的意义有了一些出入（要知道，这个字眼早在SDK时代即有了）。因此，你可能会看到有些书籍这么说：MDI 程序使用CMultiDocTemplate，SDI 程序使用CSingleDocTemplate。

Cdocument——当你为自己的程序由CDocument 派生出一个子类后，应该在其中加上成员变量，以容纳文件数据；并加上成员函数，负责修改文件内容以及

读写文件。读写文件由虚函数Serialize 负责。第８章的Scribble Step1 范例程序有极佳的示范。

Cview——此类负责将文件内容呈现到显示装置上：也许是屏幕，也许是打印机。文件内容的呈现由虚函数OnDraw负责。由于这个类实际上就是你在屏幕上所看到的窗口（外再罩一个外框窗口），所以它也负责用户输入的第一线服务。例如第８章的 Scribble Step1 范例，其View 类便处理了鼠标的按键动作。

High level abstractions

视觉性UI对象属于此类，例如工具栏CToolBar、状态栏CStatusBar、对话框列CDialogBar。加强型的View也属此类，如可卷动的ScrollView、以对话框为基础的CFormView、小型文字编辑器CEditView、树状结构的CTreeView，支持RTF文件格式的CRichEditView 等等。

Afx 全局函数

还记得吧，C++ 并不是纯种的面向对象语言（SmallTalk 和Java才是）。所以，MFC之中得以存在有不属于任何类的全局函数，它们统统在函数名称开头冠以Afx。

下面是几个常见的 Afx 全局函数：

函数名称	说明
AfxWinInit	被 *WinMain*（由 MFC 提供）呼叫的一個函式，用做 MFC GUI 程式初始化的一部份。請看第 6 章的「*AfxWinInit* - AFX 內部初始化動作」一節。如果你寫一個 MFC console 程式，就得自行呼叫此函式（請參考 Visual C++ 所附之 Tear 範例程式）。
AfxBeginThread	開始一個新的執行緒（請看第 14 章，# 756 頁）。
AfxEndThread	結束一個舊的執行緒（請看第 14 章，# 756 頁）。
AfxFormatString1	類似 *printf* 一般地將字串格式化。
AfxFormatString2	類似 *printf* 一般地將字串格式化。
AfxMessageBox	類似 Windows API 函式 *MessageBox*。
AfxOutputDebugString	將字串輸往除錯裝置（請參考附錄 D，# 924 頁）。
AfxGetApp	取得 application object（*CWinApp* 衍生物件）的指標。
AfxGetMainWnd	取得程式主視窗的指標。
AfxGetInstance	取得程式的 instance handle。
AfxRegisterClass	以自定的 *WNDCLASS* 註冊視窗類別（如果 MFC 提供的數個視窗類別不能滿足你的話）。

MFC宏（macros）

CObject和CRuntimeClass之中封装了数个所谓的object services，包括「取得运行时期的类信息」（RTTI）、Serialization（文件读写）、动态产生对象...等等。所有派生自CObject的类，都继承这些机能。我想你对这些名词及其代表的意义已经不再陌生 -- 如果你没有错过第

３章的「MFC 六大技术模拟」的话。

取得运行时期的类信息（RTTI），使你能够决定一个运行时期的对象的类信息，这样的能力在你需要对函数参数做一些额外的类型检验，或是当你要针对对象属于某种类而做特别的动作时，份外有用。

Serialization 是指将对象内容写到文件中，或从文件中读出。如此一来对象的生命就可以在程序结束之后还延续下去，而在程序重新启动之后，再被读入。

这样的对象可说是 "persistent"（永续存在）。

所谓动态的对象生成（Dynamic object creation），使你得以在运行时期产生一个特定的对象。例如document、view、和frame对象就都必须支持动态对象生成，因为framework 需要在运行时期产生它们（第８章有更详细的说明）。

此外，OLE 常常需要在运行时期做对象的动态生成动作。例如一个OLE server 程序必须能够动态产生 OLE items，用以反应 OLE client 的需求。

MFC 针对上述这些机能，准备了一些宏，让程序能够很方便地继承并实现上述四大机能。这些宏包括：

宏名称	提供机能	出现章节
DECLARE_DYNAMIC	執行時期類別資訊	第３章、第８章
IMPLEMENT_DYNAMIC	執行時期類別資訊	第３章、第８章
DECLARE_DYNCREATE	動態生成	第３章、第８章
IMPLEMENT_DYNCREATE	動態生成	第３章、第８章
DECLARE_SERIAL	物件內容的檔案讀寫	第３章、第８章
IMPLEMENT_SERIAL	物件內容的檔案讀寫	第３章、第８章
DECLARE_OLECREATE	OLE 物件的動態生成	不在本書範圍之內
IMPLEMENT_OLECREATE	OLE 物件的動態生成	不在本書範圍之內

我也已经在第３章提过MFC的消息映射（Message Mapping）与命令循环（Command Routing）两个特性。这两个性质系由以下这些 MFC 宏完成：

宏名称	提供机能	出现章节

DECLARE_MESSAGE_MAP	宣告訊息映射表資料結構	第3章、第9章
BEGIN_MESSAGE_MAP	開始訊息映射表的建置	第3章、第9章
ON_COMMAND	增加訊息映射表中的項目	第3章、第9章
ON_CONTROL	增加訊息映射表中的項目	本書未舉例
ON_MESSAGE	增加訊息映射表中的項目	???
ON_OLECMD	增加訊息映射表中的項目	本書未舉例
ON_REGISTERED_MESSAGE	增加訊息映射表中的項目	本書未舉例
ON_REGISTERED_THREAD_MESSAGE	增加訊息映射表中的項目	本書未舉例
ON_THREAD_MESSAGE	增加訊息映射表中的項目	本書未舉例
ON_UPDATE_COMMAND_UI	增加訊息映射表中的項目	第3章、第9章
END_MESSAGE_MAP	結束訊息映射表的建置	第3章、第9章

事实上，与其它MFC Programming书籍相比较，本书最大的一个特色就是，要把上述这些MFC宏的来龙去脉交待得非常清楚。我认为这对于撰写MFC程序是非常重要的一件事。

MFC 数据类型（data types）

下面所列的这些数据类型，常常出现在 MFC 之中。其中的绝大部分都和一般的Win32程序（SDK 程序）所用的相同。

下面这些是和 Win32 程序（SDK 程序）共同使用的数据类型：

```
数据类型                                 意义
BOOL                   Boolean值（布尔值，不是TRUE就是FALSE）
BSTR                   32-bit字符指针
BYTE                   8-bit整数，未带正负号
COLORREF               32-bit 数值，代表一个颜色值
DWORD                  32-bit 整数，未带正负号
LONG                   32-bit 整数，带正负号
LPARAM                 32-bit 数值，做为窗口函数或 callback 函数的一个参数
LPCSTR                 32-bit 指针，指向一个常数字符串
LPSTR                  32-bit 指针，指向一个字符串
LPCTSTR                32-bit 指针，指向一个常数字符串。此字符串可移植到Unicode和DBCS（双字节字集）
LPTSTR                 32-bit 指针，指向一个字符串。此字符串可移植到  Unicode和DBCS（双位组字集）
LPVOID                 32-bit 指针，指向一个未指定类型的数据
LPRESULT               32-bit数值，做为窗口函数或callback函数的回返值
UINT                   在Win16中是一个16-bit未带正负号整数，在Win32中是一个  32-bit未带正负号整数
WNDPROC                32-bit 指针，指向一个窗口函数
WORD                   16-bit 整数，未带正负号
WPARAM                 窗口函数的callback函数的一个参数。在Win16中是  16 bits，在Win32中是32 bit
```

下面这些是MFC独特的数据类型：

```
数据类型                          意义
POSITION      一个数值，代表 collection 对象（例如数组或串列）中的元素位置。常使用于MFC collection c
LPCRECT        32-bit指针，指向一个不变的  RECT 结构。
```

前面所说那些MFC数据类型与C++语言数据类型之间的对应，定义于 WINDEF.H中。我列出其中一部分，并且将不符合(_MSC_VER >= 800) 条件式的部分略去。

```
#define NULL 0
#define far // 侯俊杰注：Win32 不再有 far 或 near memory model,
#define near // 而是使用所谓的 flat model。pascall 函数调用习惯
#define pascal __stdcall  // 也被 stdcall 函数调用习惯取而代之。
#define cdecl _cdecl
#define CDECL _cdecl
#define CALLBACK __stdcall//侯俊杰注：在Windows programming演化过程中
#define WINAPI __stdcall  // 曾经出现的 PASCAL、CALLBACK、WINAPI、
#define WINAPIV __cdecl// APIENTRY，现在都代表相同的意义，就是 stdcall
#define APIENTRY WINAPI// 函数调用习惯。
#define APIPRIVATE __stdcall
#define PASCAL __stdcall
#define FAR far
#define NEAR near
#define CONST const
typedef unsigned long DWORD;
typedef int BOOL;
typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef float FLOAT;
typedef FLOAT *PFLOAT;
typedef BOOL near *PBOOL;
typedef BOOL far *LPBOOL;
typedef BYTE near *PBYTE;
typedef BYTE far *LPBYTE;
typedef int near *PINT;
typedef int far *LPINT;
typedef WORD near *PWORD;
typedef WORD far *LPWORD;
typedef long far *LPLONG;
typedef DWORD near *PDWORD;
typedef DWORD far *LPDWORD;
typedef void far *LPVOID;
typedef CONST void far *LPCVOID;
typedef int INT;
typedef unsigned int UINT;
typedef unsigned int *PUINT;
/* Types use for passing & returning polymorphic values */
typedef UINT WPARAM;
typedef LONG LPARAM;
typedef LONG LRESULT;
typedef DWORD COLORREF;
typedef DWORD *LPCOLORREF;
typedef struct tagRECT
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
typedef const RECT FAR* LPCRECT;
typedef struct tagPOINT
{
    LONG  x;
    LONG  y;
} POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;
typedef struct tagSIZE
{
    LONG cx;
    LONG cy;
} SIZE, *PSIZE, *LPSIZE;
```

第 6 章 MFC程序设计导论

> MFC 程序的生死因果
>
> 理想如果不向实际做点妥协，理想就会归于尘土。
>
> 中华民国还得十次革命才得建立，面向对象怎能把一切传统都抛开。

以传统的C/SDK撰写Windows 程序，最大的好处是可以清楚看见整个程序的来龙去脉和消息动向，然而这些重要的动线在MFC应用程序中却隐晦不明，因为它们被Application Framework 包起来了。这一章主要目的除了解释MFC应用程序的长像，也要从MFC原始代码中检验出一个Windows 程序原本该有的程序进入点（WinMain）、窗口类注册（RegisterClass）、窗口产生（CreateWindow）、消息循环（Message Loop）、窗口函数（Window Procedure）等等动作，抽丝剥茧彻底了解一个 MFC 程序的诞生与结束，以及生命过程。

为什么要安排这一章？了解MFC 内部构造是必要的吗？看电视需要知道映射管的原理吗？开汽车需要知道传动轴与变速箱的原理吗？学习MFC不就是要一举超越烦琐的Windows API？啊，厂商（不管是哪一家）广告给我们的印象就是，藉由可视化的工具我们可以一步登天，基本上这个论点正确，只是有个但书：你得学会操控Application Framework。

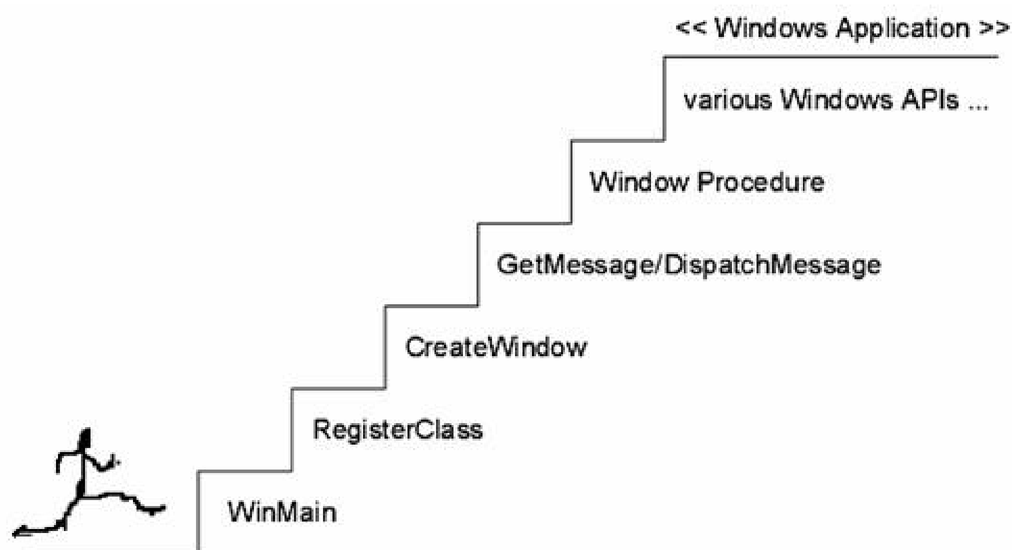
想象你拥有一部保时捷，风驰电挚风光得很，但是引擎盖打开来全傻了眼。如果你懂汽车内部运作原理，那么至少开车时「脚不要老是含着离合器，以免来令片磨损」这个道理背后的原理你就懂了，「踩煞车时绝不可以同时踩离合器，以免失去引擎煞车力」这个道理背后的原理你也懂了，甚至你的保时捷要保养维修时或也可以不假外力自己来。

不要把自己想象成这场游戏中的后座车主，事实上作为这本技术书籍的读者的你，应该是车厂师傅。

好，这个比喻不见得面面俱到，但起代码你知道了自己的身份。

题外话：我的朋友曾铭源（现在纽约工作）写信给我说：『最近项目的压力大，人员纷纷离职。接连一个多礼拜，天天有人上门面谈。人事部门不知从哪里找来这些阿哥，号称有三年的SDK/MFC经验，结果对起话来是鸡同鸭讲，WinMain 和 Windows Procedure都搞不清楚。问他什么是message handler？只会在 ClassWizard 上 click、click、click !!! 拜Wizard 之赐，人力市场上多出了好几倍的 VC/MFC 程序员，但这些「Wizard 通」我们可不敢要』。

以 raw Windows API 开发程序，学习的路径是单纯的，条理分明的，你一定先从程序进入点开始，然后产生窗口类，然后产生窗口，然后取得消息，然后分辨消息，然后决定如何处理消息。虽然动作繁琐，学习却容易。

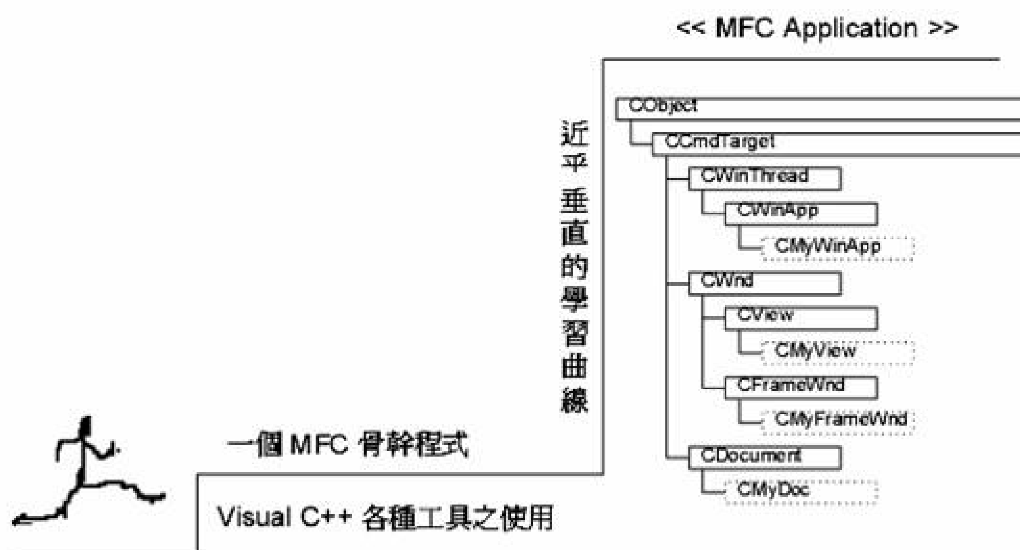


我希望你了解，本书之所以在各个主题中不厌其烦地挖MFC内部动作，解释骨干程序的每一条指令，每一个环节，是为了让你踏实地接受MFC，进而有能力役使MFC。你以为这是一条远路？呵呵，似远实近！

不二法门：熟记 MFC 类的阶层架构

MFC在1.0 版时期的诉求是「一组将SDK API包装得更好用的类库」，从 2.0 版开始更进一步诉求是一个「Application Framework」，拥有重要的 Document-View 架构；随后又在更新版本上增加了 OLE 架构、DAO 架构...。为了让你有一个最轻松的起点，我把第一个程序简化到最小程度，舍弃 Document-View架构，使你能够尽快掌握C++/MFC 程序的面貌。这个程序并不以AppWizard制作出来，也不以ClassWizard 管理维护，而是纯手工打造。毕竟 Wizards 做出来的程序代码有一大堆批注，某些批注对Wizards有特殊意义，不能随便删除，却可能会混淆初学者的视听焦点；而且 Wizards所产生的程序骨干已具备 Document-View 架构，又有许多奇奇怪怪的宏，初学者暂避为妙。我们目前最想知道的是一个最阳春的 MFC 程序以什么面貌呈现，以及它如何开始运作，如何结束生命。

以MFC开发程序，一开始很快速，因为开发工具会为你产生一个骨干程序，一般该有的各种界面一应俱全。但是 MFC的学习曲线十分陡峭，程序员从骨干程式出发一直到有能力修改程序代码以符合个人的需要，是一段不易攀登的峭壁。



如果我们了解 Windows 程序的基本运作原理，并了解 MFC 如何把这些基础动作整合起来，我们就能够使 MFC 学习曲线的陡峭程度缓和下来。因此能够迅速接受 MFC，进而使用 MFC。呵，一条似远实近的道路！



SDK 程序设计的第一要务是了解最重要的数个API函数的意义和用法，像 是RegisterClass、CreateWindow、GetMessage、DispatchMessage，以及消息的获得与分配。

MFC 程序设计的第一要务则是熟记MFC的类阶层架构，并清楚知晓其中几个一定会用到的类。本书最后面有一张MFC 4.2架构图，迭床架屋，令人畏惧，我将挑出单单两个类，组合成一个 "Hello MFC" 程序。这两个类在 MFC 的地位如图 6-1 所示。

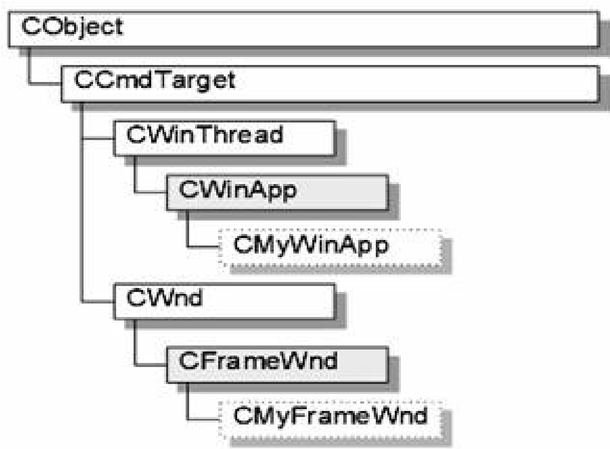


图6-1 本章范例程序所用到的MFC类

需要什么函数库？

开始写代码之前，我们得先了解程序代码以外的外围环境。第一个必须知道的是，MFC 程序需要什么函数库？SDK 程序链接时期所需的函数库已在第一章显示，MFC 程序一样需要它们：

Windows C Runtime 函数库（VC++ 5.0）

檔案名稱	檔案大小	說明
LIBC.LIB	898826	C Runtime 函式庫的靜態聯結版本
MSVCRT.LIB	510000	C Runtime 函式庫的動態聯結版本
MSVCRTD.LIB	803418	'D' 表示使用於 Debug 模式

＊这些函数库不再区分Large/Medium/Small内存模式，因为32位操作系统不再有记忆体模式之分。这些函数库的多线程版本，请参考本书#38页。

DLL Import函数库（VC++ 5.0）

檔案名稱	檔案大小	說明
GDI32.LIB	307520	for GDI32.DLL（136704 bytes in Win95）
USER32.LIB	517018	for USER32.DLL（45568 bytes in Win95）
KERNEL32.LIB	635638	for KERNEL32.DLL（413696 bytes in Win95）
...		

此外，应用程序还需要链接一个所谓的MFC函数库，或称为AFX 函数库，它也就是MFC这个application framework 的本体。你可以静态链接之，也可以动态链接之，AppWizard给你选择权。本例使用动态链接方式，所以需要一个对应的MFC import 函数库：

MFC 函数库（AFX 函数库)（VC++ 5.0，MFC 4.2）

檔案名稱	檔案大小	說明
MFC42.LIB	4200034	MFC42.DLL（941840 bytes）的 import 函式庫。
MFC42D.LIB	3003766	MFC42D.DLL（1393152 bytes）的 import 函式庫。
MFCS42.LIB	168364	
MFCS42D.LIB	169284	
MFCN42D.LIB	91134	
MFCD42D.LIB	486334	
MFCO42D.LIB	2173082	
...		

我们如何在链接器（ link.exe）中设定选项，把这些函数库都链接起来？稍后在HELLO.MAK中可以一窥全貌。

如果在Visual C++整合环境中工作，这些设定不劳你自己动手，整合环境会根据我们圈选的项目自动做出一个合适的 makefile。这些 makefile 的内容看起来非常诘屈聱牙，事实上我们也不必太在意它，因为那是整合环境的工作。这一章我不打算依赖任何开发工具，一切自己来，你会在稍后看到一个简洁清爽的makefile。

需要什么包含文件？

SDK 程序只要包含WINDOWS.H 就好，所有API的函数声明、消息定义、常数定义、宏定义、都在WINDOWS.H 文件中。除非程序另调用了操作系统提供的新模块（如CommDlg、ToolHelp、DDEML...），才需要再各别包含对应的.H文件。

WINDOWS.H 过去是一个巨大文件，大约在 5000 行上下。现在已拆分内容为数十个较小的.H 文件，再由WINDOWS.H 包含进来。也就是说它变成一个 "Master included file for Windows applications"。

MFC 程序不这么单纯，下面是它常常需要面对的另外一些.H 文件：

- STDAFX.H——这个文件用来做为Precompiled header file（请看稍后的方块说明），其内只是含入其它的MFC表头文件。应用程序通常会准备自己的STDAFX.H，例如本章的Hello程序就在STDAFX.H 中包含AFXWIN.H。
- AFXWIN.H——每一个Windows MFC 程序都必须包含它，因为它以及它所包含的文件声明了所有的MFC 类。此文件内含AFX.H，后者又包含AFXVER_.H，后者又包含AFXV_W32.H，后者又包含 WINDOWS.H（啊呼，终于现身）。
- AFXEXT.H——凡使用工具栏、状态栏之程序必须包含这个文件。

- AFXDLGS.H——凡使用通用型对话框（Common Dialog）之MFC程序需包含此文件，其内部包含COMMDLG.H。
- AFXCMN.H——凡使用Windows 95新增之通用型控制组件（Common Control） 之MFC程序需包含此文件。
- AFXCOLL.H——凡使用Collections Classes（用以处理数据结构如数组、串列）之程序必须包含此文件。
- AFXDLLX.H——凡MFC extension DLLs均需包含此文件。
- AFXRES.H——MFC 程序的RC文件必须包含此文件。MFC 对于标准资源（例如File、Edit 等菜单项目）的ID 都有默认值，定义于此文件中，例如：

```
// File commands
#define ID_FILE_NEW 0xE100
#define ID_FILE_OPEN 0xE101
#define ID_FILE_CLOSE 0xE102
#define ID_FILE_SAVE 0xE103
#define ID_FILE_SAVE_AS 0xE104
...
// Edit commands
#define ID_EDIT_COPY 0xE122
#define ID_EDIT_CUT 0xE123
...
```

这些菜单项目都有预设的说明文字（将出现在状态栏中），但说明文字并不会事先定义于此文件，AppWizard为我们制作骨干程序时才把说明文字加到应用程序的RC文件中。第 4 章的骨干程序Scribble step0的RC文件中就有这样的字符串表格：

```
STRINGTABLE DISCARDABLE
BEGIN
  ID_FILE_NEW      "Create a new document"
  ID_FILE_OPEN     "Open an existing document"
  ID_FILE_CLOSE    "Close the active document"
  ID_FILE_SAVE     "Save the active document"
  ID_FILE_SAVE_AS  "Save the active document with a new name"
  ...
  ID_EDIT_COPY "Copy the selection and puts it on the Clipboard"
  ID_EDIT_CUT  "Cut the selection and puts it on the Clipboard"
  ...
END
```

所有MFC包含文件均置于\MSVC\MFC\INCLUDE 中。这些文件连同Windows SDK的包含文件 WINDOWS.H、COMMDLG.H、TOOLHELP.H、DDEML.H... 每每在编译过程中耗费大量的时间，因此你绝对有必要设定 Precompiled header。

Precompiled Header

一个应用程序在发展过程中常需要不断地编译。Windows 程序包含的标准.H 文件非常巨大但内容不变，编译器浪费在这上面的时间非常多。Precompiled header 就是将.H 文件第一次编译后的结果贮存起来，第二次再编译时就可以直接从磁盘中取出来用。这种观念在 Borland C/C++ 早已行之，Microsoft 这边则是一直到 Visual C++ 1.0 才具备。

简化的 MFC 程序架构一以 Hello MFC 为例

现在我们正式进入 MFC 程序设计。由于 Document/View 架构复杂，不适合初学者，所以我先把它略去。这里所提的程序观念是一般的 MFC Application Framework 的子集合。

本章程序名为Hello，运行时会在窗口中从天而降"Hello, MFC"字样。Hello 是一个非常简单而具代表性的程序，它的代表性在于：

- 每一 个MFC程序都想从MFC中派生出适当的类来用（不然又何必以MFC 写程序呢），其中两个不可或缺的类 CWinApp和 CFrameWnd在 Hello程序中会表现出来，它们的意义如图 6-2。
- MFC类中某些函数一定得被应用程序改写（例如 CWinApp::InitInstance），这在Hello程序中也看得到。
- 菜单和对话框，Hello 也都具备。

图 6-3 是 Hello 源文件的组成。第一次接触MFC程序，我们常常因为不熟悉MFC的类分类、类命名规则，以至于不能在脑中形成具体印象，于是细部讨论时各种信息及说明彷如过眼烟云。相信我，你必须多看几次，并且用心熟记 MFC 命名规则。

图 6-3 之后是 Hello 程序的原始代码。由于MFC已经把Windows API都包装起来了，原始代码再也不能够「说明一切」。你会发现 MFC 程序很有点见林不见树的味道：

- 看不到WinMain，因此不知程序从哪里开始执行。
- 看不到RegisterClass 和 CreateWindow，那么窗口是如何做出来的呢？
- 看不到Message Loop（GetMessage/DispatchMessage），那么程序如何推动？
- 看不到Window Procedure，那么窗口如何运作？

我的目的就在铲除这些困惑。

Hello 程序原始代码

```
    HELLO.MAK - makefile
    RESOURCE.H - 所有资源ID 都在这里定义。本例只定义一个IDM_ABOUT。
    JJHOUR.ICO - 图标文件，用于主窗口和对话框。
   HELLO.RC - 资源描述文件。本例有一份菜单、一个图示、和一个对话框。
    STDAFX.H - 包含 AFXWIN.H。
    STDAFX.CPP  - 包含STDAFX.H，为的是制造出Precompiled header。
    HELLO.H - 声明CMyWinApp和CMyFrameWnd。
    HELLO.CPP  - 定义CMyWinApp和CMyFrameWnd。
```

注意：没有模块定义文件 .DEF？是的，如果你不指定模块定义文件，链接器就使用默认值。

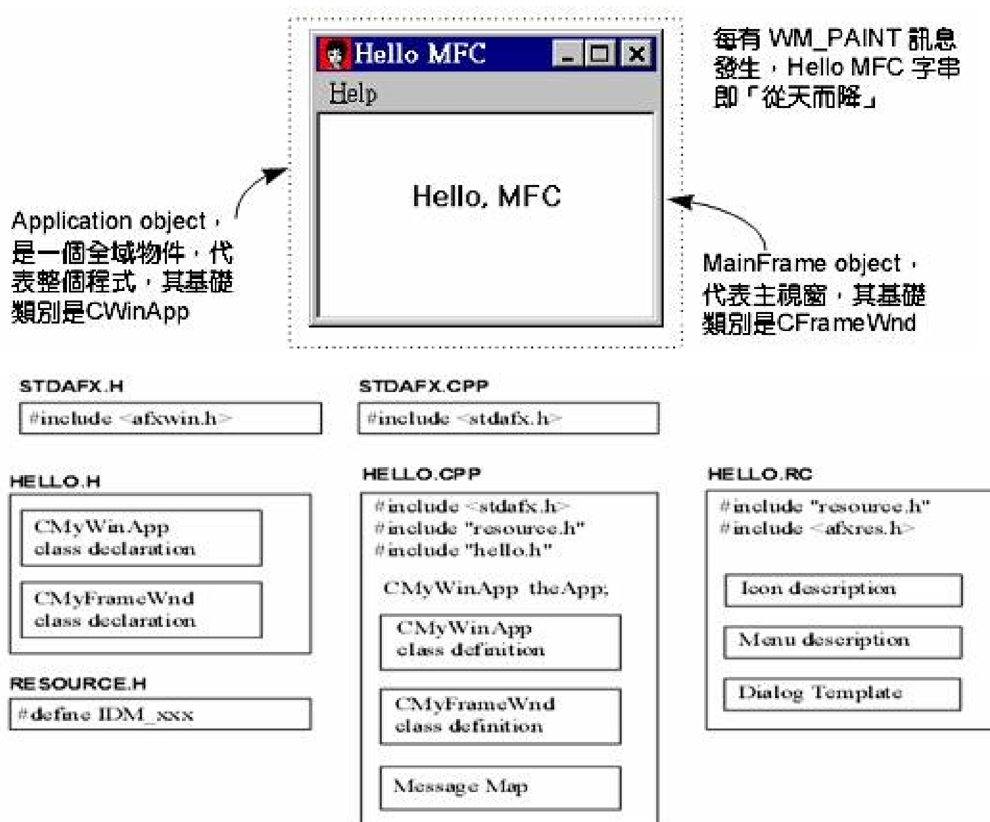


图6-3 Hello程序的基本文件架构。一般习惯为每个类准备

一个 .H（声明 ）和一个.CPP（实现），本例把两类集中在一起是为了简化。

HELLO.MAK（请在 DOS 窗口中执行 nmake hello.mak。环境设定请参考 p.224）

```
#0001  # filename : hello.mak
#0002  # make file for hello.exe (MFC 4.0 Application)
#0003  # usage : nmake hello.mak (Visual C++ 5.0)
#0004
#0005  Hello.exe : StdAfx.obj Hello.obj Hello.res
#0006    link.exe /nologo /subsystem:windows /incremental:no \
#0007             /machine:I386 /out:"Hello.exe" \
#0008             Hello.obj StdAfx.obj Hello.res \
#0009             msvcrt.lib kernel32.lib user32.lib gdi32.lib mfc42.lib
#0010
#0011  StdAfx.obj : StdAfx.cpp StdAfx.h
#0012    cl.exe /nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" \
#0013          /D "_AFXDLL" /D "_MBCS" /Fp"Hello.pch" /Yc"stdafx.h"  \
#0014          /c StdAfx.cpp
#0015
#0016  Hello.obj : Hello.cpp Hello.h StdAfx.h
#0017    cl.exe /nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" \
#0018          /D "_AFXDLL" /D "_MBCS" /Fp"Hello.pch" /Yu"stdafx.h"  \
#0019          /c Hello.cpp
#0020
#0021  Hello.res : Hello.rc Hello.ico jjhour.ico
#0022    rc.exe /l 0x404 /Fo"Hello.res" /D "NDEBUG" /D "_AFXDLL" Hello.rc
```

RESOURCE.H

```
#0001  // resource.h
#0002  #define IDM_ABOUT 100
```

HELLO.RC

```
#0001  // hello.rc
#0002  #include "resource.h"
#0003  #include "afxres.h"
#0004
#0005  JJHouRIcon        ICON  DISCARDABLE  "JJHOUR.ICO"
#0006  AFX_IDI_STD_FRAME  ICON  DISCARDABLE  "JJHOUR.ICO"
#0007
#0008  MainMenu MENU DISCARDABLE
#0009  {
#0010      POPUP "&Help"
#0011      {
#0012        MENUITEM "&About HelloMFC...", IDM_ABOUT
#0013      }
#0014  }
#0015
#0016  AboutBox DIALOG DISCARDABLE  34, 22, 147, 55
#0017  STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
#0018  CAPTION "About Hello"
#0019  {
#0020    ICON          "JJHouRIcon",IDC_STATIC,11,17,18,20
#0021    LTEXT         "Hello MFC 4.0",IDC_STATIC,40,10,52,8
#0022    LTEXT         "Copyright 1996 Top Studio",IDC_STATIC,40,25,100,8
#0023    LTEXT         "J.J.Hou",IDC_STATIC,40,40,100,8
#0024    DEFPUSHBUTTON "OK",IDOK,105,7,32,14,WS_GROUP
#0025  }
```

STDAFX.H

```
#0001 // stdafx.h : include file for standard system include files,
#0002 // or project specific include files that are used frequently,
#0003 // but are changed infrequently
#0004
#0005 #include <afxwin.h>   // MFC core and standard components
```

STDAFX.CPP

```
#0001 // stdafx.cpp : source file that includes just the standard includes
#0002 //      Hello.pch will be the pre-compiled header
#0003 //      stdafx.obj will contain the pre-compiled type information
#0004
#0005 #include "stdafx.h"
```

HELLO.H

```
#0001 //---------------------------------------------------------------
#0002 //             MFC 4.0  Hello Sample Program
#0003 //        Copyright (c) 1996   Top Studio * J.J.Hou
#0004 //
#0005 // 檔名      : hello.h
#0006 // 作者      : 侯俊傑
#0007 // 編譯聯結  : 請參考 hello.mak
#0008 //
#0009 // 宣告 Hello 程式的兩個類別 : CMyWinApp 和 CMyFrameWnd
#0010 //---------------------------------------------------------------
#0011
#0012 class CMyWinApp : public CWinApp
#0013 {
#0014 public:
#0015    BOOL InitInstance();  // 每一個應用程式都應該改寫此函式
#0016 };
#0017
#0018 //---------------------------------------------------------------
#0019 class CMyFrameWnd : public CFrameWnd
#0020 {
#0021 public:
#0022    CMyFrameWnd();            // constructor
#0023    afx_msg void OnPaint();   // for WM_PAINT
#0024    afx_msg void OnAbout();   // for WM_COMMAND (IDM_ABOUT)
#0025
#0026 private:
#0027    DECLARE_MESSAGE_MAP()     // Declare Message Map
#0028    static VOID CALLBACK LineDDACallback(int,int,LPARAM);
#0029    // 注意: callback 函式必須是 "static"，才能去除隱藏的 'this' 指標。
#0030 };
```

HELLO.CPP

```
#0001 //---------------------------------------------------------------
#0002 //               MFC 4.0  Hello sample program
#0003 //        Copyright (c) 1996   Top Studio * J.J.Hou
#0004 //
#0005 // 檔名      : hello.cpp
#0006 // 作者      : 侯俊傑
#0007 // 編譯聯結  : 請參考 hello.mak
#0008 //
#0009 // 本例示範最簡單之 MFC 應用程式，不含 Document/View 架構。程式每收到
#0010 // WM_PAINT 即利用 GDI 函式 LineDDA() 讓 "Hello, MFC" 字串從天而降。
#0011 //---------------------------------------------------------------
#0012 #include "Stdafx.h"
#0013 #include "Hello.h"
#0014 #include "Resource.h"
#0015
#0016 CMyWinApp theApp;   // application object
#0017
#0018 //---------------------------------------------------------------
#0019 // CMyWinApp's member
#0020 //---------------------------------------------------------------
#0021 BOOL CMyWinApp::InitInstance()
#0022 {
#0023   m_pMainWnd = new CMyFrameWnd();
#0024   m_pMainWnd->ShowWindow(m_nCmdShow);
#0025   m_pMainWnd->UpdateWindow();
#0026   return TRUE;
#0027 }
#0028 //---------------------------------------------------------------
#0029 // CMyFrameWnd's member
#0030 //---------------------------------------------------------------
#0031 CMyFrameWnd::CMyFrameWnd()
#0032 {
#0033   Create(NULL, "Hello MFC", WS_OVERLAPPEDWINDOW, rectDefault,
#0034          NULL, "MainMenu");     // "MainMenu" 定義於 RC 檔
#0035 }
#0036 //---------------------------------------------------------------
#0037 BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0038   ON_COMMAND(IDM_ABOUT, OnAbout)
#0039   ON_WM_PAINT()
#0040 END_MESSAGE_MAP()
#0041 //---------------------------------------------------------------
#0042 void CMyFrameWnd::OnPaint()
#0043 {
#0044 CPaintDC dc(this);
```

```
#0045 CRect rect;
#0046
#0047    GetClientRect(rect);
#0048
#0049    dc.SetTextAlign(TA_BOTTOM | TA_CENTER);
#0050
#0051    ::LineDDA(rect.right/2, 0, rect.right/2, rect.bottom/2,
#0052        (LINEDDAPROC) LineDDACallback, (LPARAM) (LPVOID) &dc);
#0053 }
#0054 //----------------------------------------------------------------
#0055 VOID CALLBACK CMyFrameWnd::LineDDACallback(int x, int y, LPARAM lpdc)
#0056 {
#0057 static char szText[] = "Hello, MFC";
#0058
#0059    ((CDC*)lpdc)->TextOut(x, y, szText, sizeof(szText)-1);
#0060    for(int i=1; i<50000; i++);  // 純粹是爲了延遲下降速度，以利觀察
#0061 }
#0062 //----------------------------------------------------------------
#0063 void CMyFrameWnd::OnAbout()
#0064 {
#0065    CDialog about("AboutBox", this);  // "AboutBox" 定義於 RC 檔
#0066    about.DoModal();
#0067 }
```

上面这些程序代码中，你看到了一些MFC类如CWinApp和CFrameWnd，一些MFC数据类型如BOOL和VOID，一些MFC宏如 DECLARE_MESSAGE_MAP和BEGIN_MESSAGE_END和END_MESSAGE_MAP。这些都曾经在第 5 章的「纵览 MFC」一节中露过脸。但是单纯从C++ 语言的角度来看，还有一些是我们不能理解的，如HELLO.H 中的afx_msg（#23 行）和CALLBACK（#28 行）。

你可以在WINDEF.H 中发现CALLBACK 的意义：

```
#define CALLBACK  __stdcall // 一种函数调用习惯
```

可以在AFXWIN.H中发现afx_msg 的意义：

```
#define afx_msg // intentional placeholder
// 故意安排的一个空位置。也许以后版本会用到。
```

MFC 程序的来龙去脉（causal relations）

让我们从第 1 章的C/SDK 观念出发，看看MFC程序如何运作。

第一件事情就是找出 MFC 程序的进入点。MFC 程序也是 Windows 程序，所以它应该也有一个WinMain，但是我们在 Hello 程序看不到它的踪影。是的，但先别急，在程序进入点之前，更有一个（而且仅有一个）全局对象（本例名为 theApp），这是所谓的application object，当操作系统将程序加载并启动，这个全局对象获得配置，其构造函数会先执行，比 WinMain 更早。所以以时间顺序来说，我们先看看这个 application object。

我只借用两个类 ：CWinApp 和 CFrameWnd

你已经看过了图 6-2，作为一个最最粗浅的MFC程序，Hello是如此单纯，只有一个窗口。回想第一章 Generic 程序的写法，其主体在于 WinMain 和 WndProc，而这两个部分其实都有相当程度的不变性。好极了，MFC 就把有着相当固定行为之 WinMain 内部动作包装在 CWinApp 中，把有着相当固定行为之 WndProc 内部动作包装在 CFrameWnd 中。也就是说：

- CWinApp 代表程序本体
- CFrameWnd 代表一个框架窗口（Frame Window）

但虽然我说，WinMain 内部动作和 WndProc 内部动作都有着相当程度的固定行为，它们毕竟需要面对不同应用程序而有某种变化。所以，你必须以这两个类为基础，派生自己的类，并改写其中一部分成员函数。

```
class CMyWinApp : public CWinApp
{
    ...
};
class CMyFrameWnd : public CFrameWnd
{
    ...
};
```

本章对派生类的命名规则是：在基类名称的前面加上"My"。这种规则真正上战场时不见得适用，大型程序可能会自同一个基类派生出许多自己的类。不过以教学目的而言，这种命名方式使我们从字面就知道类之间的从属关系，颇为理想（根据我的经验，初学者会被类的命名搞得头昏脑胀）。

CwinApp——取代 WinMain 的地位

CWinApp 的派生对象被称为application object，可以想见，CWinApp 本身就代表一个程序本体。一个程序的本体是什么？回想第 1 章的SDK 程序，与程序本身有关而不与窗口有关的数据或动作有些什么？系统传进来的四个WinMain 参 数 算 不 算 ？InitApplication 和 InitInstance 算不算？消息循环算不算？都算，是的，以下是MFC 4.x 的CWinApp声明（节录自AFXWIN.H）：

```
class CWinApp : public CWinThread
{
// Attributes
    // Startup args (do not change)
    HINSTANCE m_hInstance;
    HINSTANCE m_hPrevInstance;
    LPTSTR m_lpCmdLine;
    int m_nCmdShow;

    // Running args (can be changed in InitInstance)
    LPCTSTR m_pszAppName;  // human readable name
    LPCTSTR m_pszRegistryKey;   // used for registry entries
```

```
public:  // set in constructor to override default
    LPCTSTR m_pszExeName;      // executable name (no spaces)
    LPCTSTR m_pszHelpFilePath;  // default based on module path
    LPCTSTR m_pszProfileName;   // default based on app name

public:
    // hooks for your initialization code
    virtual BOOL InitApplication();

    // overrides for implementation
    virtual BOOL InitInstance();
    virtual int ExitInstance();
    virtual int Run();
    virtual BOOL OnIdle(LONG lCount);
...
};
```

几乎可以说CWinApp 用来取代WinMain在SDK程序中的地位。这并不是说MFC程序没有WinMain（稍后我会解释），而是说传统上 SDK 程序的 WinMain 所完成的工作现在由CWinApp 的三个函数完成：

```
virtual BOOL InitApplication();
virtual BOOL InitInstance();
virtual int  Run();
```

WinMain 只是扮演役使它们的角色。

会不会觉得CWinApp 的成员变量中少了点什么东西？是不是应该有个成员变量记录主窗口的handle（或是主窗口对应之C++ 对象）？的确，在MFC 2.5 中的确有m_pMainWnd 这么个成员变量（以下节录自MFC 2.5 的AFXWIN.H）：

```
class CWinApp : public CCmdTarget
{
// Attributes
    // Startup args (do not change)
    HINSTANCE m_hInstance;
    HINSTANCE m_hPrevInstance;
    LPSTR m_lpCmdLine;
    int m_nCmdShow;

    // Running args (can be changed in InitInstance)
    CWnd* m_pMainWnd;           // main window (optional)
    CWnd* m_pActiveWnd;         // active main window (may not be m_pMainWnd)
    const char* m_pszAppName;   // human readable name

public:  // set in constructor to override default
    const char* m_pszExeName;      // executable name (no spaces)
    const char* m_pszHelpFilePath;  // default based on module path
    const char* m_pszProfileName;   // default based on app name

public:
    // hooks for your initialization code
    virtual BOOL InitApplication();
    virtual BOOL InitInstance();

    // running and idle processing
    virtual int Run();
    virtual BOOL OnIdle(LONG lCount);

    // exiting
    virtual int ExitInstance();
...
};
```

但从MFC 4.x开始，m_pMainWnd 已经被移往CWinThread 中了（它是 CWinApp 的父类）。以下内容节录自 MFC 4.x 的 AFXWIN.H：

```
class CWinThread : public CCmdTarget
{
// Attributes
    CWnd* m_pMainWnd;      // main window (usually same AfxGetApp()->m_pMainWnd)
    CWnd* m_pActiveWnd;   // active main window (may not be m_pMainWnd)

    // only valid while running
    HANDLE m_hThread;       // this thread's HANDLE
    DWORD m_nThreadID;      // this thread's ID

    int GetThreadPriority();
    BOOL SetThreadPriority(int nPriority);

// Operations
    DWORD SuspendThread();
    DWORD ResumeThread();

// Overridables
    // thread initialization
    virtual BOOL InitInstance();

    // running and idle processing
    virtual int Run();
    virtual BOOL PreTranslateMessage(MSG* pMsg);
    virtual BOOL PumpMessage();     // low level message pump
    virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing

public:
    // valid after construction
    AFX_THREADPROC m_pfnThreadProc;
...
};
```

熟悉Win32的朋友，看到CWinThread 类之中的SuspendThread 和 ResumeThread成员函数，可能会发出会心微笑。

CFrameWnd －取代 WndProc 的地位

CFrameWnd 主要用来掌握一个窗口，几乎你可以说它是用来取代SDK程序中的窗口函数的地位。传统的SDK窗口函数写法是：

```
long FAR PASCAL WndProc(HWND hWnd, UNIT msg, WORD wParam, LONG lParam)
{
    switch(msg) {
        case WM_COMMAND :
            switch(wParam) {
                case IDM_ABOUT :
                    OnAbout(hWnd, wParam, lParam);
                    break;
            }
            break;
        case WM_PAINT :
            OnPaint(hWnd, wParam, lParam);
            break;
        default :
            DefWindowProc(hWnd, msg, wParam, lParam);
    }
}
```

MFC程序有新的作法，我们在Hello程序中也为CMyFrameWnd 准备了两个消息处理例程，声明如下：

```
class CMyFrameWnd : public CFrameWnd
{
public:
        CMyFrameWnd();
        afx_msg void OnPaint();
        afx_msg void OnAbout();
        DECLARE_MESSAGE_MAP()
};
```

OnPaint 处理什么消息？OnAbout又是处理什么消息？我想你很容易猜到，前者处理WM_PAINT，后者处理WM_COMMAND的IDM_ABOUT。这看起来十分利落，但让人搞不懂来龙去脉。程序中是不是应该有「把消息和处理函数关联在一起」的设定动作？是的，这些设定在HELLO.CPP才看得到。但让我先着一鞭：DECLARE_MESSAGE_MAP宏与此有关。

这种写法非常奇特，原因是 MFC 内建了一个所谓的Message Map机制，会把消息自动送到「与消息对映之特定函数」去；消息与处理函数之间的对映关系由程序员指定。

DECLARE_MESSAGE_MAP 另搭配其它宏，就可以很便利地将消息与其处理函数关联在一起：

```
BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
ON_WM_PAINT()
ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()
```

稍后我就来探讨这些神秘的宏。

引爆器－Application object

我们已经看过HELLO.H声明的两个类，现在把目光转到HELLO.CPP 身上。这个文件将两个类实现出来，并产生一个所谓的application object。故事就从这里展开。

下面这张图包括右半部的Hello原始代码与左半部的MFC原始代码。从这一节以降，我将以此图解释MFC程序的启动、运行、与结束。不同小节的图将标示出当时的程序进行状况。

HELLO.CPP

```
❶ CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
   m_pMainWnd = new CMyFrameWnd();
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
   Create(NULL, "Hello MFC", ...,
          "MainMenu");
}

void CMyFrameWnd::OnPaint()   { ... }
void CMyFrameWnd::OnAbout()   { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
   ON_COMMAND(IDM_ABOUT, OnAbout)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

上图的theApp就是Hello程序的application object，每一个MFC应用程序都有一个，而且也只有这么一个。当你执行Hello，这个全局对象产生，于是构造函数执行起来。我们并没有定义CMyWinApp 构造函数；至于其父类 CWinApp 的构造函数内容摘要如下（摘录自APPCORE.CPP）：

```
CWinApp::CWinApp(LPCTSTR lpszAppName)
{
    m_pszAppName = lpszAppName;

    // initialize CWinThread state
    AFX_MODULE_THREAD_STATE* pThreadState = AfxGetModuleThreadState();
    pThreadState->m_pCurrentWinThread = this;
    m_hThread = ::GetCurrentThread();
    m_nThreadID = ::GetCurrentThreadId();

    // initialize CWinApp state
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    pModuleState->m_pCurrentWinApp = this;

    // in non-running state until WinMain
    m_hInstance = NULL;
    m_pszHelpFilePath = NULL;
    m_pszProfileName = NULL;
    m_pszRegistryKey = NULL;
    m_pszExeName = NULL;
    m_lpCmdLine = NULL;
    m_pCmdInfo = NULL;
    ...
}
```

CWinApp之中的成员变量将因为theApp这个全局对象的诞生而获得配置与初值。如果程序中没有theApp存在，编译链接还是可以顺利通过，但运行时会出现系统错误消息。

隐晦不明的 WinMain

HELLO.CPP

```
CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
    m_pMainWnd = new CMyFrameWnd();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
    Create(NULL, "Hello MFC", ...,
           "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_COMMAND(IDM_ABOUT, OnAbout)
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

theApp 配置完成后，WinMain 登场。我们并未撰写 WinMain 程序代码，这是 MFC 早已准备好并由链接器直接加到应用程序代码中的，其原始代码列于图 6-4。_tWinMain 函数的 是为了支持Unicode而准备的一个宏。

```
// in APPMODUL.CPP
extern "C" int WINAPI
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
          LPTSTR lpCmdLine, int nCmdShow)
{
      // call shared/exported WinMain
      return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}
```

此外，在DLLMODUL.CPP中有一个DllMain函数。本书并未涵盖DLL程序设计。

```
// in WINMAIN.CPP
#0001 /////////////////////////////////////////////////////////////////
#0002 // Standard WinMain implementation
#0003 //  Can be replaced as long as 'AfxWinInit' is called first
#0004
#0005 int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
#0006         LPTSTR lpCmdLine, int nCmdShow)
#0007 {
#0008     ASSERT(hPrevInstance == NULL);
#0009
#0010     int nReturnCode = -1;
#0011     CWinApp* pApp = AfxGetApp();
#0012
#0013     // AFX internal initialization
#0014     if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
#0015             goto InitFailure;
#0016
#0017     // App global initializations (rare)
#0018     ASSERT_VALID(pApp);
#0019     if (!pApp->InitApplication())
#0020             goto InitFailure;
#0021     ASSERT_VALID(pApp);
#0022
#0023     // Perform specific initializations
#0024     if (!pApp->InitInstance())
#0025     {
#0026         if (pApp->m_pMainWnd != NULL)
#0027         {
#0028             TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
#0029             pApp->m_pMainWnd->DestroyWindow();
#0030         }
#0031         nReturnCode = pApp->ExitInstance();
#0032         goto InitFailure;
#0033     }
#0034     ASSERT_VALID(pApp);
#0035
#0036     nReturnCode = pApp->Run();
#0037     ASSERT_VALID(pApp);
#0038
#0039 InitFailure:
#0040
#0041     AfxWinTerm();
#0042     return nReturnCode;
#0043 }
```

图6-4 Windows 程序进入点。原始代码可从MFC的WINMAIN.CPP中获得

稍加整理去芜存菁，就可以看到这个「程序进入点」主要做些什么事：

```
int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
    int nReturnCode = -1;
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
    return nReturnCode;
}
```

其中，AfxGetApp是一个全局函数，定义于AFXWIN1.INL中：

```
_AFXWIN_INLINE CWinApp* AFXAPI AfxGetApp()
{ return afxCurrentWinApp; }
```

而afxCurrentWinApp 又定义于 AFXWIN.H 中：

```
#define afxCurrentWinApp  AfxGetModuleState()->m_pCurrentWinApp
```

再根据稍早所述 CWinApp::CWinApp 中的动作，我们于是知道，AfxGetApp 其实就是取得 CMyWinApp 对象指针。所以，AfxWinMain 中这样的动作：

```
CWinApp* pApp = AfxGetApp();
pApp->InitApplication();
pApp->InitInstance();
nReturnCode = pApp->Run();
```

其实就相当于调用：

```
CMyWinApp::InitApplication();
CMyWinApp::InitInstance();
CMyWinApp::Run();
```

因而导至调用：

```
CWinApp::InitApplication();//因为CMyWinApp并没有改写InitApplication
CMyWinApp::InitInstance(); //因为 CMyWinApp改写了InitInstance
CWinApp::Run();            //因为 CMyWinApp并没有改写 Run
```

根据第 1 章SDK程序设计的经验推测，InitApplication 应该是注册窗口类的场所？InitInstance 应该是产生窗口并显示窗口的场所？Run应该是攫取消息并分派消息的场所？有对有错！以下数节我将实际带你看看MFC的原始代码，如此一来就可以了解隐藏在MFC背后的玄妙了。我的终极目标并不在 MFC 原始代码（虽然那的确是学习设计一个application framework 的好教材），我只是想拿把刀子把MFC看似朦胧的内部运作来个大解剖，挑出其经脉；有这种扎实的根基，使用 MFC 才能知其然并知其所以然。下面小节分别讨论AfxWinMain的四个主要动作以及引发的行为。

AfxWinInit －AFX 内部初始化动作

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

❷   AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

HELLO.CPP

```
❶ CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
    m_pMainWnd = new CMyFrameWnd();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
    Create(NULL, "Hello MFC", ...,
           "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_COMMAND(IDM_ABOUT, OnAbout)
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

我想你已经清楚看到了，AfxWinInit 是继 CWinApp 构造函数之后的第一个动作。以下是它的动作摘要（节录自 APPINIT.CPP）：

```
BOOL AFXAPI AfxWinInit(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
    ASSERT(hPrevInstance == NULL);

    // set resource handles
    AFX_MODULE_STATE* pState = AfxGetModuleState();
    pState->m_hCurrentInstanceHandle = hInstance;
    pState->m_hCurrentResourceHandle = hInstance;

    // fill in the initial state for the application
    CWinApp* pApp = AfxGetApp();
    if (pApp != NULL)
    {
        // Windows specific initialization (not done if no CWinApp)

        pApp->m_hInstance = hInstance;
        pApp->m_hPrevInstance = hPrevInstance;
        pApp->m_lpCmdLine = lpCmdLine;
        pApp->m_nCmdShow = nCmdShow;
        pApp->SetCurrentHandles();
    }

    // initialize thread specific data (for main thread)
    if (!afxContextIsDLL)
        AfxInitThread();

    return TRUE;
}
```

其中调用的AfxInitThread函数的动作摘要如下（节录自 THRDCORE.CPP）：

```
void AFXAPI AfxInitThread()
{
    if (!afxContextIsDLL)
    {
        // attempt to make the message queue bigger
        for (int cMsg = 96; !SetMessageQueue(cMsg) && (cMsg -= 8); )
            ;

        // set message filter proc
        _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
        ASSERT(pThreadState->m_hHookOldMsgFilter == NULL);
        pThreadState->m_hHookOldMsgFilter = ::SetWindowsHookEx(WH_MSGFILTER,
            _AfxMsgFilterHook, NULL, ::GetCurrentThreadId());

        // intialize CTL3D for this thread
        _AFX_CTL3D_STATE* pCtl3dState = _afxCtl3dState;
        if (pCtl3dState->m_pfnAutoSubclass != NULL)
            (*pCtl3dState->m_pfnAutoSubclass)(AfxGetInstanceHandle());

        // allocate thread local _AFX_CTL3D_THREAD just for automatic termination
        _AFX_CTL3D_THREAD* pTemp = _afxCtl3dThread;
    }
}
```

如果你曾经看过本书前身Visual C++ 面向对象MFC程序设计，我想你可能对这句话印象深刻：「WinMain 一开始即调用AfxWinInit，注册四个窗口类」。这是一个已成昨日黄花的事实。MFC 的确会为我们注册四个窗口类，但不再是在AfxWinInit 中完成。稍后我会把注册动作挖出来，那将是窗口诞生前一刻的行为。

CWinApp::InitApplication

HELLO.CPP

```
❶ CMyWinApp theApp;    // application object

BOOL CMyWinApp::InitInstance()
{
    m_pMainWnd = new CMyFrameWnd();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
    Create(NULL, "Hello MFC", ...,
           "MainMenu");
}

void CMyFrameWnd::OnPaint()   { ... }
void CMyFrameWnd::OnAbout()   { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_COMMAND(IDM_ABOUT, OnAbout)
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

❷   AfxWinInit(...);

❸   pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

AfxWinInit之后的动作是pApp->InitApplication。稍早我说过了，pApp 指向CMyWinApp对象（也就是本例的theApp），所以，当程序调用：

```
pApp->InitApplication();
```

相当于调用：

```
CMyWinApp::InitApplication();
```

但是你要知道，CMyWinApp继承自 CWinApp，而InitApplication又是 CWinApp 的一个虚函数；我们并没有改写它（大部分情况下不需改写它），所以上述动作相当于调用：

```
CWinApp::InitApplication();
```

此函数之原始代码出现在APPCORE.CPP 中：

```
BOOL CWinApp::InitApplication()
{
    if (CDocManager::pStaticDocManager != NULL)
    {
        if (m_pDocManager == NULL)
            m_pDocManager = CDocManager::pStaticDocManager;
        CDocManager::pStaticDocManager = NULL;
    }

    if (m_pDocManager != NULL)
        m_pDocManager->AddDocTemplate(NULL);
    else
        CDocManager::bStaticInit = FALSE;

    return TRUE;
}
```

这些动作都是MFC为了内部管理而做的。

关于Document Template和CDocManager，第 7 章和第 8 章另有说明。

CMyWinApp::InitInstance

HELLO.CPP

```
❶ CMyWinApp theApp;    // application object

BOOL CMyWinApp::InitInstance()
{
    m_pMainWnd = new CMyFrameWnd();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
    Create(NULL, "Hello MFC", ...,
           "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_COMMAND(IDM_ABOUT, OnAbout)
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

❷   AfxWinInit(...);

❸   pApp->InitApplication();
❹   pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

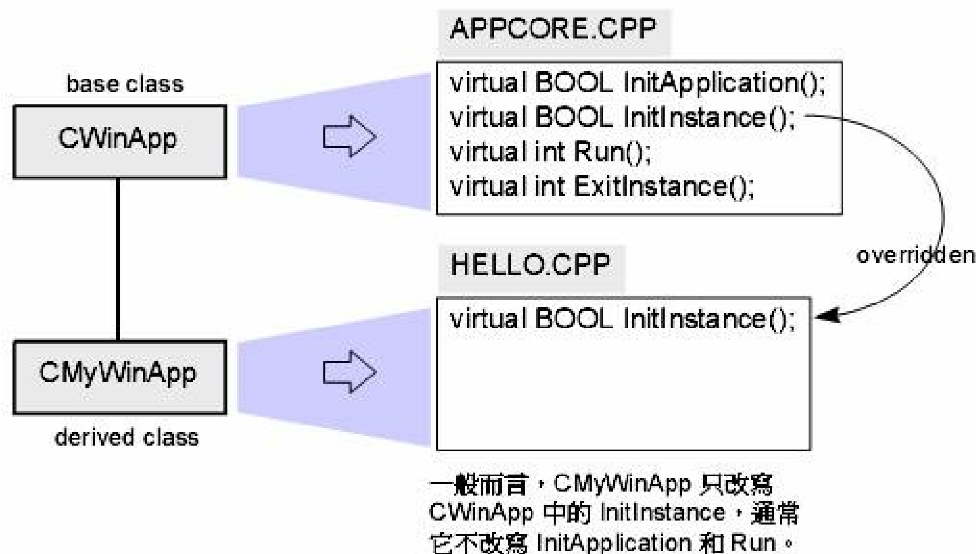
继InitApplication 之后，AfxWinMain调用 pApp->InitInstance。稍早我说过了，pApp 指向 CMyWinApp 对象（也就是本例的 theApp），所以，当程序调用：

```
pApp->InitInstance();
```

相当于调用

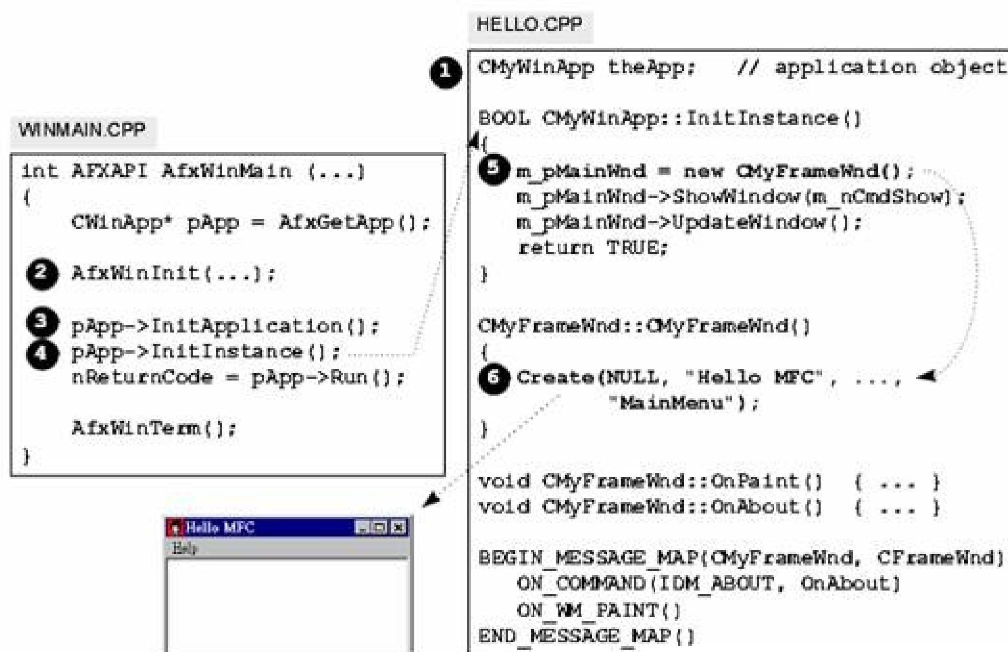
```
CMyWinApp::InitInstance();
```

但是你要知道，CMyWinApp继承自CWinApp，而InitInstance 又是CWinApp 的一个虚函数。由于我们改写了它，所以上述动作的的确确就是调用我们自己（CMyWinApp）的这个 InitInstance 函数。我们将在该处展开我们的主窗口生命。



注意：应用程序一定要改写虚函数 InitInstance，因为它在 CWinApp 中只是个空函数，没有任何内建（预设）动作。

CFrameWnd::Create 产生主窗口（并先注册窗口类）



CMyWinApp::InitInstance 一开始new 了一个CMyFrameWnd 对象，准备用作框架窗口的C++ 对象。new 会引发构造函数：

```
CMyFrameWnd::CMyFrameWnd
{
    Create(NULL, "Hello MFC", WS_OVERLAPPEDWINDOW, rectDefault, NULL, "MainMenu");
}
```

其中Create是CFrameWnd 的成员函数，它将产生一个窗口。但，使用哪一个窗口类呢？

这里所谓的「窗口类」是由 RegisterClass 所注册的一份数据结构，不是 C++ 类。

根据 CFrameWnd::Create 的规格：

```
BOOL Create( LPCTSTR lpszClassName,
             LPCTSTR lpszWindowName,
             DWORD dwStyle = WS_OVERLAPPEDWINDOW,
             const RECT& rect = rectDefault,
             CWnd* pParentWnd = NULL,
             LPCTSTR lpszMenuName = NULL,
             DWORD dwExStyle = 0,
             CCreateContext* pContext = NULL );
```

八个参数中的后六个参数都有默认值，只有前两个参数必须指定。第一个参 数 lpszClassName指定WNDCLASS 窗口类，我们放置NULL 究竟代表什么意思？意思是要以MFC内建的窗口类产生一个标准的外框窗口。但，此时此刻Hello程序中根本不存在任何窗口类呀！噢，Create 函数在产生窗口之前会引发窗口类的注册动作，稍后再解释。

第三个参数 dwStyle 指定窗口风格，预设是 WS_OVERLAPPEDWINDOW，也正是最常用的一种，它 被定义为（在 WINDOWS.H 之中）：

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED | WS_CAPTION |
                             WS_SYSMENU | WS_THICKFRAME |
                             WS_MINIMIZEBOX | WS_MAXIMIZEBOX)
```

因此如果你不想要窗口右上角的极大极小钮，就得这么做：

```
Create(NULL,
       "Hello MFC",
       WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME
       rectDefault,
       NULL,
       "MainMenu");
```

如果你希望窗口有垂直滚动条，就得在第三个参数上再加增WS_VSCROLL风格。第二个参数lpszWindowName指定窗口标题，本例指定 "Hello MFC"。第三除了上述标准的窗口风格，另有所谓的扩充风格，可以在Create 的第七个参数dwExStyle指定之。扩充风格唯有以::CreateWindowEx（而非::CreateWindow）函数才能完成。事实上稍后你就会发现，CFrameWnd::Create 最终调用的正是::CreateWindowEx。Windows 3.1 提供五种窗口扩充风格：

```
WS_EX_DLGMODALFRAME
WS_EX_NOPARENTNOTIFY
WS_EX_TOPMOST
WS_EX_ACCEPTFILES
WS_EX_TRANSPARENT
```

Windows 95有更多选择，包括 WS_EX_WINDOWEDGE和WS_EX_CLIENTEDGE，让窗口更具3D立体感。Framework已经自动为我们指定了这两个扩充风格。

Create 的第四个参数rect指定窗口的位置与大小。默认值rectDefault是CFrameWnd的一个static 成员变量，告诉Windows以预设方式指定窗口位置与大小，就好像在SDK 程序中以CW_USEDEFAULT 指定给CreateWindow 函数一样。如果你很有主见，

希望窗口在特定位置有特定大小，可以这么做：

```
Create(NULL,
       "Hello MFC",
       WS_OVERLAPPEDWINDOW,
       CRect(40, 60, 240, 460),  // 起始位置 (40,60)，寬 200，高 400)
       NULL,
       "MainMenu");
```

第五个参数pParentWnd 指定父窗口。对于一个top-level 窗口而言，此值应为NULL，表示没有父窗口（其实是有的，父窗口就是 desktop 窗口）。

第六个参数lpszMenuName指定选单。本例使用一份在RC中准备好的选单MainMenu。第八个参数利用它，在具备 Document/View 架构的程序中初始化外框窗口（第 8 章的「CDocTemplate管理CDocument /CView/CFrameWnd」一节中将谈到此一主题）。本例不具备Document/View 架构，所以不必指定 pContext 参数，默认值为 NULL。

第八个参数 pContext 是一个指向 CCreateContext 结构的指针，framework前面提过，CFrameWnd::Create 在产生窗口之前，会先引发窗口类的注册动作。让我再扮一次 MFC 向导，带你寻幽访胜。你会看到 MFC 为我们注册的窗口类名称，及注册动作。

WINFRM.CPP

```
BOOL CFrameWnd::Create(LPCTSTR lpszClassName,
                       LPCTSTR lpszWindowName,
                       DWORD dwStyle,
                       const RECT& rect,
                       CWnd* pParentWnd,
                       LPCTSTR lpszMenuName,
                       DWORD dwExStyle,
                       CCreateContext* pContext)
{
    HMENU hMenu = NULL;
    if (lpszMenuName != NULL)
    {
        // load in a menu that will get destroyed when window gets destroyed
        HINSTANCE hInst = AfxFindResourceHandle(lpszMenuName, RT_MENU);
        hMenu = ::LoadMenu(hInst, lpszMenuName);
    }

    m_strTitle = lpszWindowName;    // save title for later

    CreateEx(dwExStyle, lpszClassName, lpszWindowName, dwStyle,
        rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top,
        pParentWnd->GetSafeHwnd(), hMenu, (LPVOID)pContext);

    return TRUE;
}
```

函数中调用 CreateEx。注意，CWnd有成员函数CreateEx，但其派生类 CFrameWnd 并无，所以这里虽然调用的是CFrameWnd::CreateEx，其实乃是从父类继承下来的CWnd::CreateEx。

WINCORE.CPP

```
BOOL CWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
                    LPCTSTR lpszWindowName, DWORD dwStyle,
                    int x, int y, int nWidth, int nHeight,
                    HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam)
{
    // allow modification of several common create parameters
    CREATESTRUCT cs;
    cs.dwExStyle = dwExStyle;
    cs.lpszClass = lpszClassName;

    cs.lpszName = lpszWindowName;
    cs.style = dwStyle;
    cs.x = x;
    cs.y = y;
    cs.cx = nWidth;
    cs.cy = nHeight;
    cs.hwndParent = hWndParent;
    cs.hMenu = nIDorHMenu;
    cs.hInstance = AfxGetInstanceHandle();
    cs.lpCreateParams = lpParam;

    PreCreateWindow(cs);
    AfxHookWindowCreate(this);  //此動作將在第9章探討。
    HWND hWnd = ::CreateWindowEx(cs.dwExStyle, cs.lpszClass,
                 cs.lpszName, cs.style, cs.x, cs.y, cs.cx, cs.cy,
                 cs.hwndParent, cs.hMenu, cs.hInstance, cs.lpCreateParams);
    ...
}
```

函数中调用的PreCreateWindow 是虚函数，CWnd 和CFrameWnd之中都有定义。由于this指针所指对象的缘故，这里应该调用的是 CFrameWnd::PreCreateWindow（还记得第 2 章我说过虚函数常见的那种行为模式吗？）

WINFRM.CPP

```
// CFrameWnd second phase creation
BOOL CFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        cs.lpszClass = _afxWndFrameOrView;  // COLOR_WINDOW background
    }
    ...
}
```

其中AfxDeferRegisterClass是一个定义于AFXIMPL.H中的宏。

AFXIMPL.H

```
#define AfxDeferRegisterClass(fClass) \
  ((afxRegisteredClasses & fClass) ? TRUE : AfxEndDeferRegisterClass(fClass))
```

这个宏表示，如果变量 afxRegisteredClasses 的值显示系统已经注册了 fClass 这种窗口类，MFC就啥也不做；否则就调用 AfxEndDeferRegisterClass(fClass)，准备注册之。afxRegisteredClasses 定义于AFXWIN.H，是一个旗标变量，用来记录已经注册了哪些窗口类：

```
// in AFXWIN.H
#define afxRegisteredClasses  AfxGetModuleState()->m_fRegisteredClasses
```

WINCORE.CPP :

```
#0001  BOOL AFXAPI AfxEndDeferRegisterClass(short fClass)
#0002  {
#0003      BOOL bResult = FALSE;
#0004
#0005      // common initialization
#0006      WNDCLASS wndcls;
#0007      memset(&wndcls, 0, sizeof(WNDCLASS));   // start with NULL defaults
#0008      wndcls.lpfnWndProc = DefWindowProc;
#0009      wndcls.hInstance = AfxGetInstanceHandle();
#0010      wndcls.hCursor = afxData.hcurArrow;
#0011
#0012      AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
#0013      if (fClass & AFX_WND_REG)
#0014      {
#0015          // Child windows - no brush, no icon, safest default class styles
#0016          wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0017          wndcls.lpszClassName = _afxWnd;
#0018          bResult =  AfxRegisterClass(&wndcls);
#0019          if (bResult)
#0020              pModuleState->m_fRegisteredClasses |= AFX_WND_REG;
#0021      }
#0022      else if (fClass & AFX_WNDOLECONTROL_REG)
#0023      {
#0024          // OLE Control windows - use parent DC for speed
#0025          wndcls.style |= CS_PARENTDC | CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0026          wndcls.lpszClassName = _afxWndOleControl;
#0027          bResult = AfxRegisterClass(&wndcls);
#0028          if (bResult)
```

```
#0029            pModuleState->m_fRegisteredClasses |= AFX_WNDOLECONTROL_REG;
#0030     }
#0031     else if (fClass & AFX_WNDCONTROLBAR_REG)
#0032     {
#0033        // Control bar windows
#0034        wndcls.style = 0;   // control bars don't handle double click
#0035        wndcls.lpszClassName = _afxWndControlBar;
#0036        wndcls.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
#0037        bResult = AfxRegisterClass(&wndcls);
#0038        if (bResult)
#0039           pModuleState->m_fRegisteredClasses |= AFX_WNDCONTROLBAR_REG;
#0040     }
#0041     else if (fClass & AFX_WNDMDIFRAME_REG)
#0042     {
#0043        // MDI Frame window (also used for splitter window)
#0044        wndcls.style = CS_DBLCLKS;
#0045        wndcls.hbrBackground = NULL;
#0046        bResult = RegisterWithIcon(&wndcls, _afxWndMDIFrame,
                                        AFX_IDI_STD_MDIFRAME);
#0047        if (bResult)
#0048           pModuleState->m_fRegisteredClasses |= AFX_WNDMDIFRAME_REG;
#0049     }
#0050     else if (fClass & AFX_WNDFRAMEORVIEW_REG)
#0051     {
#0052        // SDI Frame or MDI Child windows or views - normal colors
#0053        wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0054        wndcls.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
#0055        bResult = RegisterWithIcon(&wndcls, _afxWndFrameOrView,
                                        AFX_IDI_STD_FRAME);
#0056        if (bResult)
#0057           pModuleState->m_fRegisteredClasses |= AFX_WNDFRAMEORVIEW_REG;
#0058     }
#0059     else if (fClass & AFX_WNDCOMMCTLS_REG)
#0060     {
#0061         InitCommonControls();
#0062         bResult = TRUE;
#0063         pModuleState->m_fRegisteredClasses |= AFX_WNDCOMMCTLS_REG;
#0064     }
#0065
#0066     return bResult;
#0067  }
```

出现在上述函数中的六个窗口类卷标代码，分别定义于AFXIMPL.H 中：

```
#define AFX_WND_REG              (0x0001)
#define AFX_WNDCONTROLBAR_REG    (0x0002)
#define AFX_WNDMDIFRAME_REG      (0x0004)
#define AFX_WNDFRAMEORVIEW_REG   (0x0008)
#define AFX_WNDCOMMCTLS_REG      (0x0010)
#define AFX_WNDOLECONTROL_REG    (0x0020)
```

出现在上述函数中的五个窗口类名称，分别定义于WINCORE.CPP中：

```
const TCHAR _afxWnd[] = AFX_WND;
const TCHAR _afxWndControlBar[] = AFX_WNDCONTROLBAR;
const TCHAR _afxWndMDIFrame[] = AFX_WNDMDIFRAME;
const TCHAR _afxWndFrameOrView[] = AFX_WNDFRAMEORVIEW;
const TCHAR _afxWndOleControl[] = AFX_WNDOLECONTROL;
```

而等号右手边的那些AFX_ 常数又定义于AFXIMPL.H 中：

```
#ifndef _UNICODE
#define _UNICODE_SUFFIX
#else
#define _UNICODE_SUFFIX _T("u")
#endif

#ifndef _DEBUG
#define _DEBUG_SUFFIX
#else
#define _DEBUG_SUFFIX _T("d")
#endif

#ifdef _AFXDLL
#define _STATIC_SUFFIX
#else
#define _STATIC_SUFFIX _T("s")
#endif

#define AFX_WNDCLASS(s) \
  _T("Afx") _T(s) _T("42") _STATIC_SUFFIX _UNICODE_SUFFIX _DEBUG_SUFFIX

#define AFX_WND                AFX_WNDCLASS("Wnd")
#define AFX_WNDCONTROLBAR      AFX_WNDCLASS("ControlBar")
#define AFX_WNDMDIFRAME        AFX_WNDCLASS("MDIFrame")
#define AFX_WNDFRAMEORVIEW     AFX_WNDCLASS("FrameOrView")
#define AFX_WNDOLECONTROL      AFX_WNDCLASS("OleControl")
```

所以，如果在 Windows 95（non-Unicode）中使用 MFC 动态链接版和除错版，五个窗口类的名称将是：

```
"AfxWnd42d"
"AfxControlBar42d"
"AfxMDIFrame42d"
"AfxFrameOrView42d"
"AfxOleControl42d"
```

如果在Windows NT（Unicode 环境）中使用 MFC静态链接版和除错版，五个窗口类的名称将是：

```
"AfxWnd42sud"
"AfxControlBar42sud"
"AfxMDIFrame42sud"
"AfxFrameOrView42sud"
"AfxOleControl42sud"
```

这五个窗口类的使用时机为何？稍后再来一探究竟。

让我们再回顾 AfxEndDeferRegisterClass 的动作。它调用两个函数完成实际的窗口类注册动作，一个是 RegisterWithIcon，一个是AfxRegisterClass：

```
static BOOL AFXAPI RegisterWithIcon(WNDCLASS* pWndCls,
        LPCTSTR lpszClassName, UINT nIDIcon)
{
    pWndCls->lpszClassName = lpszClassName;
    HINSTANCE hInst = AfxFindResourceHandle(
            MAKEINTRESOURCE(nIDIcon), RT_GROUP_ICON);
    if ((pWndCls->hIcon = ::LoadIcon(hInst, MAKEINTRESOURCE(nIDIcon))) == NULL)
    {
        // use default icon
        pWndCls->hIcon = ::LoadIcon(NULL, IDI_APPLICATION);
    }
    return AfxRegisterClass(pWndCls);
}
                ↓
BOOL AFXAPI AfxRegisterClass(WNDCLASS* lpWndClass)
{
    WNDCLASS wndcls;

    if (GetClassInfo(lpWndClass->hInstance,
                     lpWndClass->lpszClassName, &wndcls))
    {
        // class already registered
        return TRUE;
    }

    ::RegisterClass(lpWndClass);
    ...
    return TRUE;
}
```

注意，不同类的 PreCreateWindow 成员函数都是在窗口产生之前一刻被调用，准备用来注册窗口类。如果我们指定的窗口类是 NULL，那么就使用系统预设类。从 CWnd及其各个派生类的PreCreateWindow 成员函数可以看出，整个Framework 针对不同功能的窗口使用了哪些窗口类：

```
// in WINCORE.CPP
BOOL CWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WND_REG);
        ...
        cs.lpszClass = _afxWnd;    (這表示 CWnd 使用的視窗類別是 _afxWnd)
    }
    return TRUE;
}
```

```
// in WINFRM.CPP
BOOL CFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        ...
        cs.lpszClass = _afxWndFrameOrView;  (這表示 CFrameWnd 使用的視窗
    }                                        類別是 _afxWndFrameOrView)
    ...
}

// in WINMDI.CPP
BOOL CMDIFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDMDIFRAME_REG);
        ...
        cs.lpszClass = _afxWndMDIFrame;   (這表示 CMDIFrameWnd 使用的視窗
    }                                       類別是 _afxWndMDIFrame)
    return TRUE;
}

// in WINMDI.CPP
BOOL CMDIChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    ...
    return CFrameWnd::PreCreateWindow(cs);    (這表示 CMDIChildWnd 使用的視窗
}                                              類別是 _afxWndFrameOrView)

// in VIEWCORE.CPP
BOOL CView::PreCreateWindow(CREATESTRUCT & cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        ...
        cs.lpszClass = _afxWndFrameOrView;  (這表示 CView 使用的視窗
    }                                         類別是 _afxWndFrameOrView)
    ...
}
```

题外话：「Create 是一个比较粗糙的函数，不提供我们对图标（icon）或鼠标光标的设定，所以在 Create 函数中我们看不到相关参数」。这样的说法对吗？虽然「不能够让我们指定窗口图标以及鼠标光标」是事实，但这本来就与 Create 无关。回忆SDK程序，指定图标和光标形状实为RegisterClass 的责任而非CreateWindow 的责任！

MFC 程序的RegisterClass 动作并非由程序员自己来做，因此似乎难以改变图示。不过，MFC 还是开放了一个窗口，我们可以在HELLO.RC这么设定图示：

```
AFX_IDI_STD_FRAME   ICON   DISCARDABLE   "HELLO.ICO"
```

你可以从AfxEndDeferRegisterClass的第55 行看出，当它调用 RegisterWithIcon时，指定的icon 正是AFX_IDI_STD_FRAME。

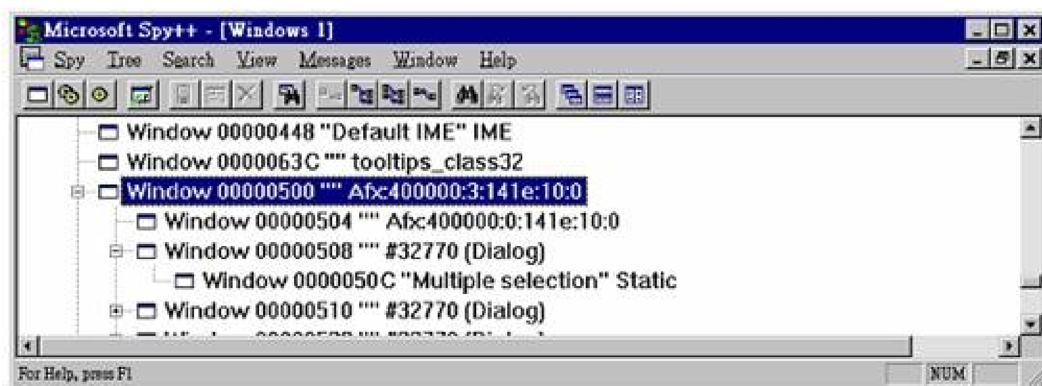
鼠标光标的设定就比较麻烦了。要改变光标形状，我们必须调用 AfxRegisterWndClass（其中有¨Cursor¨参数）注册自己的窗口类；然后再将其传回值（一个字符串）做为 Create 的第一个参数。

奇怪的窗口类名称 Afx:b:14ae:6:3e8f

当应用程序调用 CFrameWnd::Create（或 CMDIFrameWnd::LoadFrame，第 7 章）准备产生窗口时，MFC 才会在Create或LoadFrame内部所调用的 PreCreateWindow 虚函数中为你产生适当的窗口类。你已经在上一节看到了，这些窗口类的名称分别是（假设在Win95中使用MFC 4.2动态链接版和除错版）：

```
"AfxWnd42d"
"AfxControlBar42d"
"AfxMDIFrame42d"
"AfxFrameOrView42d"
"AfxOleControl42d"
```

然而，当我们以Spy++（VC++ 所附的一个工具）观察窗口类的名称，却发现：



窗口类名称怎么会变成像Afx:b:14ae:6:3e8f这副奇怪模样呢？原来是 Application Framework 玩了一些把戏，它把这些窗口类名称转换为 Afx:x:y:z:w 的型式，成为独一无二的窗口类名称：

```
x: 窗口风格（window style）的hex值

y: 窗口鼠标光标的hex值

z: 窗口背景颜色的hex值

w: 窗口图标（icon）的hex值
```

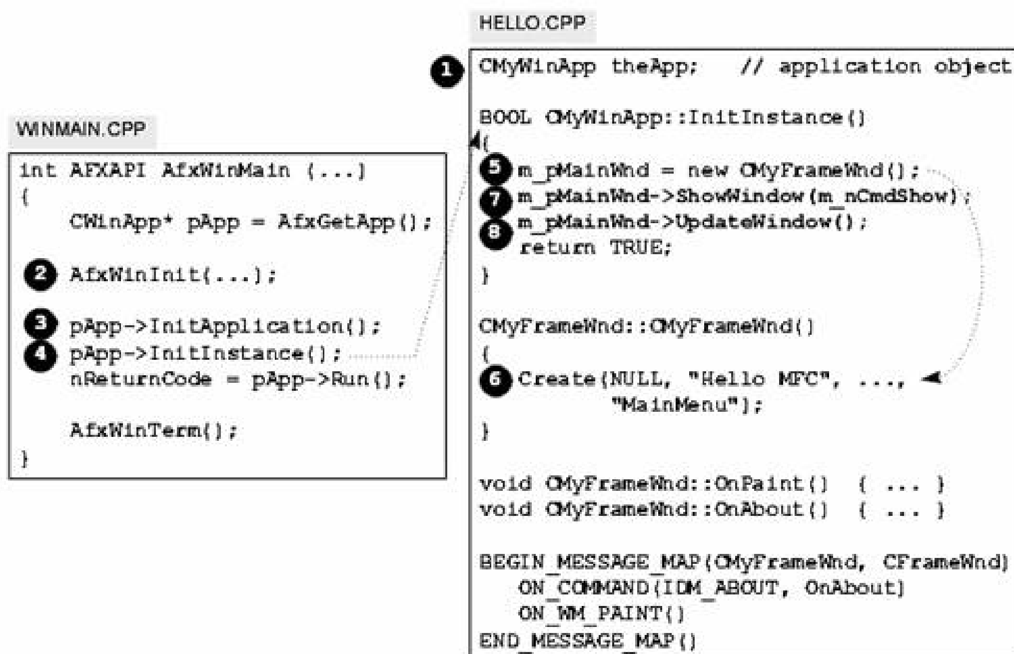
如果你要使用原来的（MFC 预设的）那些个窗口类，但又希望拥有自己定义的一个有意义的类名称，你可以改写PreCreateWindow 虚函数（因为Create 和LoadFrame的内部都会调用它），在其中先利用 API 函数 GetClassInfo 获得该类的一个副本，更改其类结构中的

lpszClassName 栏位（甚至更改其 hIcon栏位），再以AfxRegisterClass 重新注册之，例如：

```
#0000  #define MY_CLASSNAME  "MyClassName"
#0001
#0002  BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
#0003  {
#0004      static LPCSTR className = NULL;
#0005
#0006      if (!CFrameWnd::PreCreateWindow(cs))
#0007          return FALSE;
#0008
#0009      if (className==NULL) {
#0010          // One-time class registration
#0011          // The only purpose is to make the class name something
#0012          // meaningful instead of "Afx:0x4d:27:32:hup1hup:hike!"
#0013          //
#0014          WNDCLASS wndcls;
#0015          ::GetClassInfo(AfxGetInstanceHandle(), cs.lpszClass, &wndcls);
#0016          wndcls.lpszClassName = MY_CLASSNAME;
#0017          wndcls.hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
#0018          VERIFY(AfxRegisterClass(&wndcls));
#0019          className=TRACEWND_CLASSNAME;
#0020      }
#0021      cs.lpszClass = className;
#0022
#0023      return TRUE;
#0024  }
```

本书附录 D「以MFC重建Debug Window（DBWIN）」会运用到这个技巧。

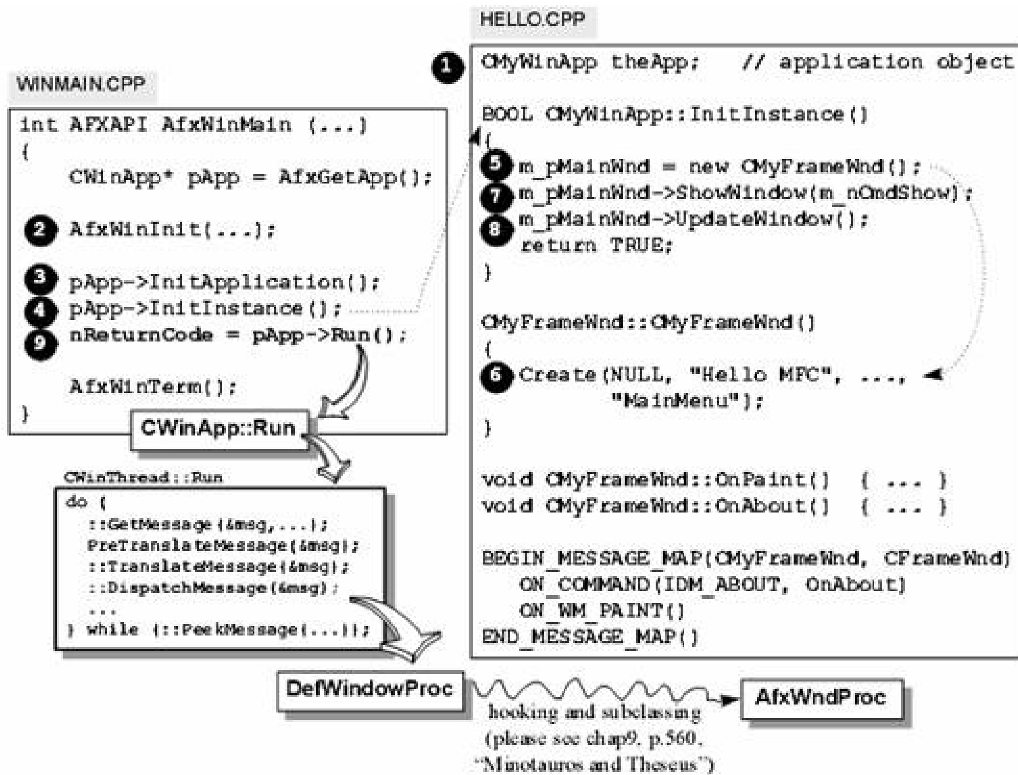
窗口显示与更新



CMyFrameWnd::CMyFrameWnd 结束后，窗口已经诞生出来；程序流程又回到 CMyWinApp::InitInstance，于是调用ShowWindow 函数令窗口显示出来，并调用 UpdateWindow 函数令 Hello 程序送出 WM_PAINT 消息。

我们很关心这个 WM_PAINT 消息如何送到窗口函数的手中。而且，窗口函数又在哪里？MFC程序是不是也像SDK 程序一样，有一个 GetMessage/DispatchMesage 循环？是否每个窗口也都有一个窗口函数，并以某种方式进行消息的判断与处理？两者都是肯定的。我们马上来寻找证据。

CWinApp::Run - 程序生命的活水源头



Hello 程序进行到这里，窗口类注册好了，窗口诞生并显示出来了，UpdateWindow 被调用，使得消息队列中出现了一个 WM_PAINT 消息，等待被处理。现在，执行的脚步到达 pApp->Run。

稍早我说过了，pApp 指向 CMyWinApp 对象（也就是本例的 theApp），所以，当程序调用：

```
pApp->Run();
```

相当于调用：

```
CMyWinApp::Run();
```

要知道，CMyWinApp 继承自CWinApp，而 Run又是CWinApp 的一个虚函数。我们并没有改写它（大部分情况下不需改写它），所以上述动作相当于调用：

```
CWinApp::Run();
```

其原始代码出现在 APPCORE.CPP 中：

```
int CWinApp::Run()
{
    if (m_pMainWnd == NULL && AfxOleGetUserCtrl())
    {
        // Not launched /Embedding or /Automation, but has no main window!
        TRACE0("Warning: m_pMainWnd is NULL in CWinApp::Run - quitting
                application.\n");
        AfxPostQuitMessage(0);
    }
    return CWinThread::Run();
}
```

32位MFC与16位MFC的巨大差异在于CWinApp与CCmdTarget之间多出了一个CWinThread，事情变得稍微复杂一些。CWinThread 定义于THRDCORE.CPP：

```
int CWinThread::Run()
{
    // for tracking the idle time state
    BOOL bIdle = TRUE;
    LONG lIdleCount = 0;

    // acquire and dispatch messages until a WM_QUIT message is received.
    for (;;)
    {
        // phase1: check to see if we can do idle work
        while (bIdle &&
                !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
        {
            // call OnIdle while in bIdle state
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }

        // phase2: pump messages while available
        do
        {
            // pump message, but quit on WM_QUIT
            if (!PumpMessage())
                return ExitInstance();

            // reset "no idle" state after pumping "normal" message
            if (IsIdleMessage(&m_msgCur))
            {
                bIdle = TRUE;
                lIdleCount = 0;
            }

        } while (::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));
    }

    ASSERT(FALSE);  // not reachable
}

BOOL CWinThread::PumpMessage()
```

```
{
    if (!::GetMessage(&m_msgCur, NULL, NULL, NULL))
    {
        return FALSE;
    }

    // process this message
    if (m_msgCur.message != WM_KICKIDLE && !PreTranslateMessage(&m_msgCur))
    {
        ::TranslateMessage(&m_msgCur);
        ::DispatchMessage(&m_msgCur);
    }
    return TRUE;
}
```

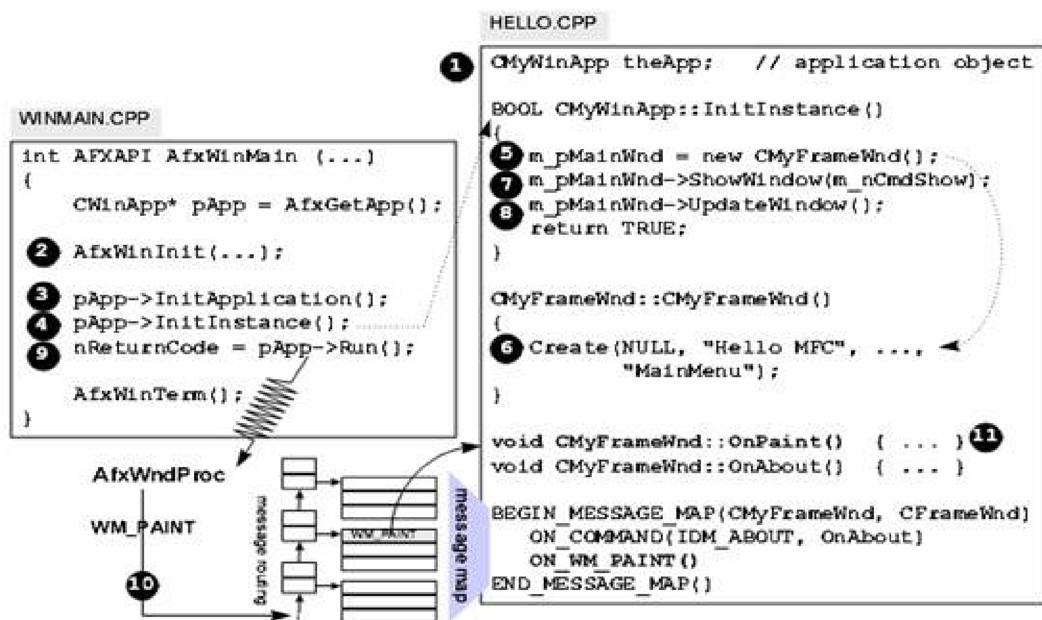
获得的消息如何交给适当的例程去处理呢？SDK 程序的作法是调用 DispatchMessage，把消息丢给窗口函数；MFC 也是如此。但我们并未在 Hello 程序中提供任何窗口函数，是的，窗口函数事实上由 MFC 提供。回头看看前面 AfxEndDeferRegisterClass 原始代码，它在注册四种窗口类之前已经指定窗口函数为：

```
wndcls.lpfnWndProc = DefWindowProc;
```

注意， 虽然窗口函数被指定为DefWindowProc成员函数，但事实上消息并不是被唧往该处，而是一个名为AfxWndProc的全局函数去。这其中牵扯到MFC暗中做了大挪移的手脚（利用hook和subclassing），我将在第 9 章详细讨论这个「乾坤大挪移」。

你看，WinMain 已由 MFC 提供，窗口类已由 MFC 注册完成、连窗口函数也都由 MFC提供。那么我们（程序员）如何为特定的消息设计特定的处理例程？MFC 应用程序对消息的辨识与判别是采用所谓的「Message Map 机制」。

把消息与处理函数串接在一起：**Message Map** 机制



基本上Message Map机制是为了提供更方便的程序接口（例如宏或表格），让程序员很方便就可以建立起消息与处理例程的对应关系。这并不是什么新发明，我在第 1 章示范了一种风格简明的 SDK 程序写法，就已经展现出这种精神。MFC 提供给应用程序使用的「很方便的接口」是两组宏。以 Hello 的主窗口为例，第一个动作是在HELLO.H 的CMyFrameWnd 加上DECLARE_MESSAGE_MAP：

```
class CMyFrameWnd : public CFrameWnd
{
public:
        CMyFrameWnd();
        afx_msg void OnPaint();
        afx_msg void OnAbout();
        DECLARE_MESSAGE_MAP()
};
```

第二个动作是在HELLO.CPP的任何位置（当然不能在函数之内）使用宏如下：

```
BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
ON_WM_PAINT()
ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()
```

这么一来就把消息WM_PAINT导到OnPaint函数，把WM_COMMAND（IDM_ABOUT）导到OnAbout 函数去了。但是，单凭一个ON_WM_PAINT 宏，没有任何参数，如何使WM_PAINT 流到 OnPaint 函数呢？

MFC 把消息主要分为三大类，Message Map 机制中对于消息与函数间的对映关系也明定以下三种：

标准Windows 消息（WM_xxx）的对映规则：

巨集名稱	對映訊息	訊息處理函式（名稱已由系統預設）
ON_WM_CHAR	WM_CHAR	OnChar
ON_WM_CLOSE	WM_CLOSE	OnClose
ON_WM_CREATE	WM_CREATE	OnCreate
ON_WM_DESTROY	WM_DESTROY	OnDestroy
ON_WM_LBUTTONDOWN	WM_LBUTTONDOWN	OnLButtonDown
ON_WM_LBUTTONUP	WM_LBUTTONUP	OnLButtonUp
ON_WM_MOUSEMOVE	WM_MOUSEMOVE	OnMouseMove
ON_WM_PAINT	WM_PAINT	OnPaint
...		

命令消息（WM_COMMAND）的一般性对映规则是：

```
ON_COMMAND(<id>,<memberFxn>)
```

例如：

```
ON_COMMAND(IDM_ABOUT,    OnAbout)
ON_COMMAND(IDM_FILENEW,  OnFileNew)
ON_COMMAND(IDM_FILEOPEN, OnFileOpen)
ON_COMMAND(IDM_FILESAVE, OnFileSave)
```

「Notification 消息」（由控制组件产生，例如BN_xxx）的对映机制的宏分为好几种（因为控制组件本就分为好几种），以下各举一例做代表：

控制元件	巨集名稱	訊息處理函式
Button	ON_BN_CLICKED(<id>,<memberFxn>)	memberFxn
ComboBox	ON_CBN_DBLCLK(<id>,<memberFxn>)	memberFxn
Edit	ON_EN_SETFOCUS(<id>,<memberFxn>)	memberFxn
ListBox	ON_LBN_DBLCLK(<id>,<memberFxn>)	memberFxn

各个消息处理函数均应以 afx_msg void 为函数型式。

为什么经过这样的宏之后，消息就会自动流往指定的函数去呢？谜底在于 Message Map的结构设计。如果你把第 3 章的Message Map仿真程序好好研究过，现在应该已是成竹在胸。我将在第 9 章再讨论MFC的Message Map。

好奇心摆两旁，还是先把实用上的问题放中间吧。如果某个消息在Message Map中找不到对映记录，消息何去何从？答案是它会往基类流窜，这个消息流窜动作称为「Message Routing」。如果一直窜到最基础的类仍找不到对映的处理例程，自会有预设函数来处理，就像 SDK 中的 DefWindowProc 一样。

MFC的CCmdTarget所派生下来的每一个类都可以设定自己的Message Map，因为它们都可能（可以）收到消息。

消息流动是个颇为复杂的机制，它和 Document/View、动态生成（Dynamic Creation），文件读写（Serialization）一样，都是需要特别留心的地方。

来龙去脉总整理

前面各节的目的就是如何将表面上看来不知所以然的MFC程序对映到我们在SDK程序设计中学习到的消息流动观念，从而清楚地掌握MFC程序的诞生与死亡。让我对MFC程序的来龙去脉再做一次总整理。

程序的诞生：

- Application object 产生，内存于是获得配置，初值亦设立了。
- AfxWinMain 执行AfxWinInit，后者又调用AfxInitThread，把消息队列尽量加大到96。
- AfxWinMain执行InitApplication。这是CWinApp 的虚函数，但我们通常不改写它。

- AfxWinMain执行InitInstance。这是CWinApp的虚函数，我们必须改写它。
- CMyWinApp::InitInstance 'new' 了一个CMyFrameWnd 对象。
- CMyFrameWnd 构造函数调用Create，产生主窗口。我们在Create 参数中指定的窗口类是NULL， 于是MFC根据窗口种类， 自行为我们注册一个名为"AfxFrameOrView42d" 的窗口类。
- 回到InitInstance中继续执行ShowWindow，显示窗口。
- 执行UpdateWindow，于是发出WM_PAINT。
- 回到AfxWinMain，执行Run，进入消息循环。

程序开始运作：

- 程序获得WM_PAINT 消息（藉由CWinApp::Run 中的::GetMessage 循环）。
- WM_PAINT经由::DispatchMessage送到窗口函数CWnd::DefWindowProc 中。
- CWnd::DefWindowProc 将消息循环过消息映射表格（Message Map）。
- 循环过程中发现有吻合项目，于是调用项目中对应的函数。此函数是应用程序利用BEGIN_MESSAGE_MAP和END_MESSAGE_MAP之间的宏设立起来的。
- 标准消息的处理例程亦有标准命名，例如WM_PAINT必然由OnPaint处理。

以下是程序的死亡：

- 用户选按【File/Close】，于是发出WM_CLOSE。
- CMyFrameWnd 并没有设置 WM_CLOSE 处理例程，于是交给预设之处理例程。
- 预设函数对于WM_CLOSE 的处 理方 式是调用::DestroyWindow，并因而发 出WM_DESTROY。
- 预设之WM_DESTROY处理方式是调用::PostQuitMessage，因此发出 WM_QUIT。
- CWinApp::Run 收到WM_QUIT后会结束其内部之消息循环，然后调用ExitInstance，这是CWinApp 的一个虚函数。
- 如果CMyWinApp 改写了ExitInstance ， 那么CWinApp::Run所调用的就是CMyWinApp::ExitInstance，否则就是CWinApp::ExitInstance。
- 最后回到AfxWinMain，执行AfxWinTerm，结束程序。

Callback 函数

Hello的OnPaint在程序收到 WM_PAINT之后开始运作。为了让"Hello, MFC" 字样从天而降并有动画效果，程序采用LineDDA API 函数。我的目的一方面是为了示范消息的处理，一方面也为了示范 MFC 程序如何调用 Windows API 函数。许多人可能不熟悉LineDDA，所以我也一并介绍这个有趣的函数。

首先介绍 LineDDA：

```
void WINAPI LineDDA(int, int, int, int, LINEDDAPROC, LPARAM);
```

这个函数用来做动画十分方便，你可以利用前四个参数指定屏幕上任意两点的(x,y) 座标，此函数将以 Bresenham 算法（注) 计算出通过两点之直线中的每一个屏幕图素座标；每计算出一个坐标，就通知由LineDDA 第五个参数所指定的 callback 函数。这个callback 函数的型式必须是：

```
typedef void (CALLBACK* LINEDDAPROC)(int, int, LPARAM);
```

通常我们在这个 callback 函数中设计绘图动作。玩过Windows的接龙游戏吗？接龙成功后扑克牌的跳动效果就可以利用LineDDA 完成。虽然扑克牌的跳动路径是一条曲线，但将曲线拆成数条直线并不困难。LineDDA 的第六个（最后一个）参数可以视应用程序的需要传递一个32位指针，本例中Hello传的是一个Device Context。

Bresenham 算法是计算机图学中为了「显示器（屏幕或打印机）系由图素构成」的这个特性而设计出来的算法，使得求直线各点的过程中全部以整数来运算，因而大幅提升计算速度。

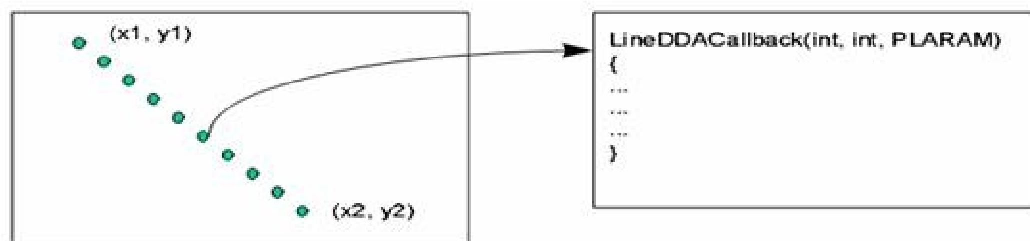


图6-6 LineDDA函数说明

你可以指定两个坐标点，LineDDA将以Bresenham 算法计算出通过两点之直线中每一个屏幕图素的坐标。每计算出一个坐标，就以该坐标为参数，调用你所指定的callback函数。

LineDDA 并不属于任何一个MFC 类，因此调用它必须使用C++ 的 "scope operator" （也就是 ::）：

```
void CMyFrameWnd::OnPaint()
{
CPaintDC dc(this);
CRect rect;

   GetClientRect(rect);

   dc.SetTextAlign(TA_BOTTOM | TA_CENTER);

   ::LineDDA(rect.right/2, 0, rect.right/2, rect.bottom/2,
       (LINEDDAPROC) LineDDACallback, (LPARAM) (LPVOID) &dc);
}
```

其中LineDDACallback是我们准备的callback 函数，必须在类中先有声明：

```
class CMyFrameWnd : public CFrameWnd
{
...
private:
   static VOID CALLBACK LineDDACallback(int,int,LPARAM);
};
```

请注意，如果类的成员函数是一个callback 函数，你必须声明它为 "static"，才能把C++ 编译器加诸于函数的一个隐藏参数this去掉（请看方块批注)。

以类的成员函数作为Windows callback函数

虽然现在来讲这个题目，对初学者而言恐怕是过于艰深，但我想毕竟还是个好机会--- 我可以在介绍如何使用callback 函数的场合，顺便介绍一些C++的重要观念。

首先我要很快地解释一下什么是callback 函数。凡是由你设计而却由 Windows系统调用的函数，统称为callback函数。这些函数都有一定的类型，以配合Windows的调用动作。

某些Windows API函数会要求以callback 函数作为其参数之一，这些API 例如SetTimer、LineDDA、EnumObjects。通常这种 API 会在进行某种行为之后或满足某种状态之时调用该callback 函数。图 6-6 已解释过 LineDDA调用 callback 函数的时机；下面即将示范的EnumObjects 则是在发现某个 Device Context 的 GDI object 符合我们的指定类型时，调用callback 函数。

好，现在我们要讨论的是，什么函数有资格在 C++ 程序中做为 callback 函数？这个问题的背后是：C++ 程序中的 callback 函数有什么特别的吗？为什么要特别提出讨论？

是的，特别之处在于，C++ 编译器为类成员函数多准备了一个隐藏参数（程序代码中看不到），这使得函数类型与 Windows callback 函数的预设类型不符。

假设我们有一个CMyclass 如下：

```
class CMyclass {
  private :
    int nCount;
    int CALLBACK _export
        EnumObjectsProc(LPSTR lpLogObject, LPSTR lpData);
  public :
    void enumIt(CDC& dc);
}
void CMyclass::enumIt(CDC& dc)
{
  // 註冊 callback 函式
  dc.EnumObjects(OBJ_BRUSH, EnumObjectsProc, NULL);
}
```

C++ 编译器针对CMyclass::enumIt实际做出来的代码相当于：

```
void CMyclass::enumIt(CDC& dc)
{
  // 註冊 callback 函式
  CDC::EnumObjects(OBJ_BRUSH, EnumObjectsProc,
                   NULL, (CDC *)&dc);
}
```

你所看到的最后一个参数，(CDC *)&dc，其实就是this指针。类成员函数靠着this指针才得以抓到正确对象的数据。你要知道，内存中只会有一份类成员函数，但却可能有许多份类成员变量——每个对象拥有一份。

C++ 以隐晦的this指针指出正确的对象。当你这么做：

```
nCount = 0;
```

其实是：

```
this->nCount = 0;
```

基于相同的道理，上例中的 EnumObjectsProc 既然是一个成员函数，C++ 编译器也会为它多准备一个隐藏参数。

好，问题就出在这个隐藏参数。callback函数是给Windows调用用的，Windows 并不经由任何对象调用这个函数，也就无由传递this指针给callback 函数，于是导至堆栈中有一个随机变量会成为this指针，而其结果当然是程序的崩溃了。

要把某个函数用作callback函数，就必须告诉C++编译器，不要放this指针作为该函数的最后一个参数。两个方法可以做到这一点：

1. 不要使用类的成员函数（也就是说，要使用全局函数）做为callback 函数。

2. 使用static 成员函数。也就是在函数前面加上static修饰词。

第一种作法相当于在 C 语言中使用callback 函数。第二种作法比较接近 OO的精神。

我想更进一步提醒你的是，C++中的static成员函数特性是，即使对象还没有产生，static 成员也已经存在（函数或变量都如此)。换句话说对象还没有产生之前你已经可以调用类的static函数或使用类的static变量了。请参阅第二章。

也就是说，凡声明为static 的东西（不管函数或变量）都并不和对象结合在一起，它们是类的一部分，不属于对象。

闲置时间（idle time）的处理：OnIdle

为了让Hello 程序更具体而微地表现一个MFC应用程序的水平，我打算为它加上闲置时间（idle time）的处理。

我已经在第 1 章介绍过了闲置时间，也简介了Win32程序如何以 PeekMessage「偷闲」。Microsoft 业已把这个观念落实到CWinApp（不，应该是CWinThread）中。请你回头看看本章的稍早的「CWinApp::Run - 程序生命的活水源头」一节，那一节已经揭露了MFC消息循环的秘密：

```
int CWinThread::Run()
{
    ...
    for (;;)
    {
        while (bIdle &&
                !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
        {
            // call OnIdle while in bIdle state
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }
        ... // msg loop
    }
}
```

CThread::OnIdle 做些什么事情呢？CWinApp 改写了OnIdle 函数，CWinApp::OnIdle 又做些什么事情呢？你可以从THRDCORE.CPP和 APPCORE.CPP 中找到这两个函数的原始代码，原始代码可以说明一切。当然基本上我们可以猜测OnIdle 函数中大概是做一些系统（指的是MFC 本身）的维护工作。这一部分的功能可以说日趋式微，因为低优先权的线程可以替代其角色。

如果你的MFC 程序也想处理idle time，只要改写CWinApp 派生类的 OnIdle 函数即可。这个函数的类型如下：

```
virtual BOOL OnIdle(LONG lCount);
```

lCount是系统传进来的一个值，表示自从上次有消息进来，到现在，OnIdle 已经被调用了多少次。稍后我将改写Hello 程序，把这个值输出到窗口上，你就可以知道闲置时间是多么地频繁。lCount会持续累增，直到 CWinThread::Run 的消息循环又获得了一个消息，此值才重置为0。

注意：Jeff Prosise 在他的Programming Windows 95 with MFC一书第 7 章谈到OnIdle函数时，曾经说过有几个消息并不会重置lCount为0，包括鼠标消息、WM_SYSTIMER、WM_PAINT。不过根据我实测的结果，至少鼠标消息是会的。稍后你可在新版的Hello程序移动鼠标，看看lCount会不会重设为0。

我如何改写Hello 呢？下面是几个步骤：

1. 在CMyWinApp中增加OnIdle 函数的声明：

```
class CMyWinApp : public CWinApp
{
public:
   virtual BOOL InitInstance();      // 每一個應用程式都應該改寫此函式
   virtual BOOL OnIdle(LONG lCount);  // OnIdle 用來處理閒置時間 (idle time)
};
```

2. 在CMyFrameWnd中增加一个IdleTimeHandler函数声明。这么做是因为我希望在窗口中显示lCount值，所以最好的作法就是在OnIdle中调用CMyFrameWnd 成员函数，这样才容易获得绘图所需的DC。

```
class CMyFrameWnd : public CFrameWnd
{
public:
   CMyFrameWnd();            // constructor
   afx_msg void OnPaint();   // for WM_PAINT
   afx_msg void OnAbout();   // for WM_COMMAND (IDM_ABOUT)
   void IdleTimeHandler(LONG lCount);  // we want it call by CMyWinApp::OnIdle
   ...
};
```

3. 在HELLO.CPP 中定义CMyWinApp::OnIdle 函数如下：

```
BOOL CMyWinApp::OnIdle(LONG lCount)
{
    CMyFrameWnd* pWnd = (CMyFrameWnd*)m_pMainWnd;
    pWnd->IdleTimeHandler(lCount);
    return TRUE;
}
```

4. 在HELLO.CPP 中定义CMyFrameWnd::IdleTimeHandler 函数如下：

```
void CMyFrameWnd::IdleTimeHandler(LONG lCount)
{
    CString str;
    CRect rect(10,10,200,30);
    CDC* pDC = new CClientDC(this);
    str.Format("%010d", lCount);
    pDC->DrawText(str, &rect, DT_LEFT | DT_TOP);
}
```

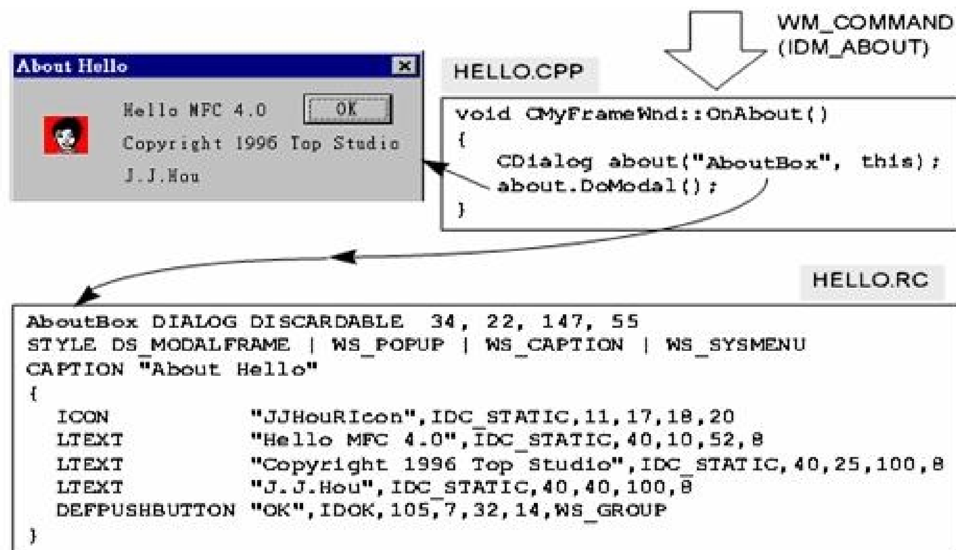
为了输出lCount，我又动用了三个MFC 类：CString、CRect 和 CDC。前两者非常简单，只是字符串与四方形结构的一层C++包装而且，后者是在 Windows 系统中绘图所必须的DC（Device Context）的一个包装。

新版Hello执行结果如下。左上角的lCount以飞快的速度更迭。移动鼠标看看，看lCount 会不会重置为0。



Dialog 与 Control

回忆SDK程序中的对话框作法：RC 文件中要准备一个对话框的Template，C 程序中要设计一个对话框函数。MFC 提供的CDialog已经把对话框的窗口函数设计好了，因此在MFC 程序中使用对话框非常地简单：



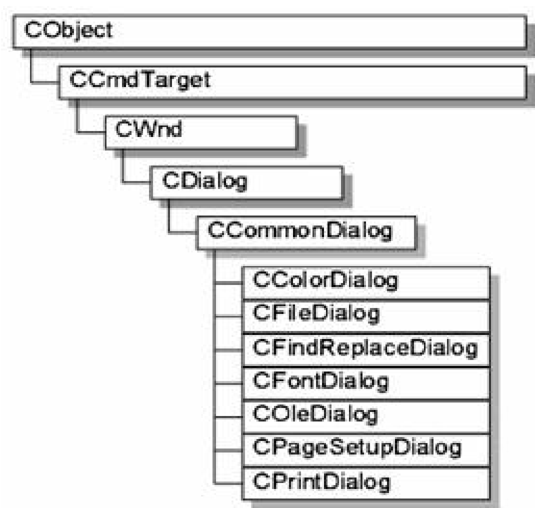
当使用者按下【File/About】选单， 根据Message Map的设定，WM_COMMAND（IDM_ABOUT）被送到OnAbout函数去。我们首先在OnAbout中产生一个CDialog 对象，名为about。CDialog 构造函数容许两个参数，第一个参数是对话框的面板资源，第二个参数是about对象的主人。由于我们的"About" 对话框是如此地简单，不需要改写CDialog 中的对话框函数，所以接下来直接调用CDialog::DoModal，对话框就开始运作了。

通用对话框（Common Dialogs）

有些对话框，例如【File Open】或【Save As】对话框，出现在每一个程序中的频率是如此之高，使微软公司不得不面对此一事实。于是，自从Windows 3.1 之后，Windows API多了一组通用对话框（Common Dialogs）API函数，系统也 多了一个对应的COMMDLG.DLL（32 位版则为COMDLG32.DLL）。

MFC 也支持通用对话框，下面是其类与其类型：

類別	型態
CCommonDialog	以下各類別的父類別
CFileDialog	File 對話盒（Open 或 Save As）
CPrintDialog	Print 對話盒
CFindReplaceDialog	Find and Replace 對話盒
CColorDialog	Color 對話盒
CFontDialog	Font 對話盒
CPageSetupDialog	Page Setup 對話盒（MFC 4.0 新增）
COleDialog	Ole 相關對話盒



在C/SDK 程序中，使用通用对话框的方式是，首先填充一块特定的结构如

OPENFILENAME，然后调用API函数如GetOpenFileName。当函数回返，结构中的某些字段便持有了用户输入的值。

MFC 通用对话框类，使用之简易性亦不输 Windows API。下面这段代码可以启动【Open】对话框并最后获得文件完整路径：

```
char szFileters[] = "Text fiels (*.txt)|*.txt|All files (*.*)|*.*||"

CFileDialog opendlg (TRUE, "txt", "*.txt",
                     OFN_FILEMUSTEXIST | OFN_HIDEREADONLY, szFilters, this);

if (opendlg.DoModal() == IDOK) {
    filename = opendlg.GetPathName();
}
```

opendlg构造函数的第一个参数被指定为TRUE，表示我们要的是一个【Open】对话框而不是【Save As】对话框。第二参数"txt"指定预设扩展名；如果用户输入的文件没有扩展名，就自动加上此一扩展名。第三个参数"*.txt" 出现在一开始的【file name】字段中。OFN_ 参数指定文件的属性。第五个参数 szFilters 指定用户可以选择的文件类型，最后一个参数是父窗口。

当DoModal回返，我们可以利用 CFileDialog的成员函数GetPathName 取得完整的文件路径。也可以使用另一个成员函数 GetFileName 取其不含路径的文件名称，或GetFileTitle 取得既不含路径亦不含扩展名的文件名称。

这便是 MFC 通用对话框类的使用。你几乎不必再从其中派生出子类，直接用就好了。

本章回顾

乍看MFC应用程序代码，实在很难推想程序的进行。一开始是一个派生自 CWinApp 的全局对象application object，然后是一个隐藏的WinMain函数，调用 application object 的 InitInstance 函数，将程序初始化。初始化动作包括构造一个窗口对象（CFrameWnd 对象），而其构造函数又调用 CFrameWnd::Create 产生真正的窗口（并在产生之前要求MFC注册窗口类）。窗口产生后 WinMain 又调用Run启动消息循环，将 WM_COMMAND（IDM_ABOUT）和WM_PAINT分别交给成员函数OnAbout和OnPaint处理。

虽然刨根究底不易，但是我们都同意，MFC 应用程序代码的确比SDK应用程序代码精简许多。事实上，MFC并不打算让应用程序代码比较容易理解，毕竟 raw Windows API才是最直接了当的动作。许许多多细碎动作被包装在MFC类之中，降低了你写程序的负担，当然，这必须建立在一个事实之上：你永远可以改变MFC的预设行为。这一点是无庸置疑的，因为所有你可能需要改变的性质，都被设计为MFC 类中的虚函数了，你可以从MFC派生出自己的类，并改写那些虚函数。

MFC 的好处在更精巧更复杂的应用程序中显露无遗。至于复杂如OLE者，那就更是非MFC不为功了。本章的Hello程序还欠缺许多Windows程序完整功能，但它毕竟是一个好起点，有点晦涩但不太难。下一章范例将运用MDI、Document/View、各式各样的UI对象...。

第 7 章 简单而完整：MFC骨干程序

> 当技术愈来愈复杂，
>
> 入门愈来愈困难，
>
> 我们的困惑愈来愈深，
>
> 犹豫愈来愈多。

上一章的Hello范例，对于MFC 程序设计导入很适合。但它只发挥了MFC的一小部份特性，只用了三个MFC类（CWinApp、CFrameWnd 和CDialog）。这一章我们要看一个完整的MFC应用程序骨干（注），其中包括丰富的UI对象（如工具栏、状态列）的生成，以及很重要的Document/View 架构观念。

注：我所谓的MFC应用程序骨干，指的是由AppWizard 产生出来的MFC程序，也就是像第 4 章所产生的 Scribble step0 那样的程序。

不二法门：熟记 MFC 类阶层架构

我还是要重复这一句话：MFC 程序设计的第一要务是熟记各类的阶层架构，并清楚了解其中几个一定会用到的类。一个MFC骨干程序（不含 ODBC 或 OLE 支持）运用到的类如图 7-1 所示，请与图6-1做个比较。

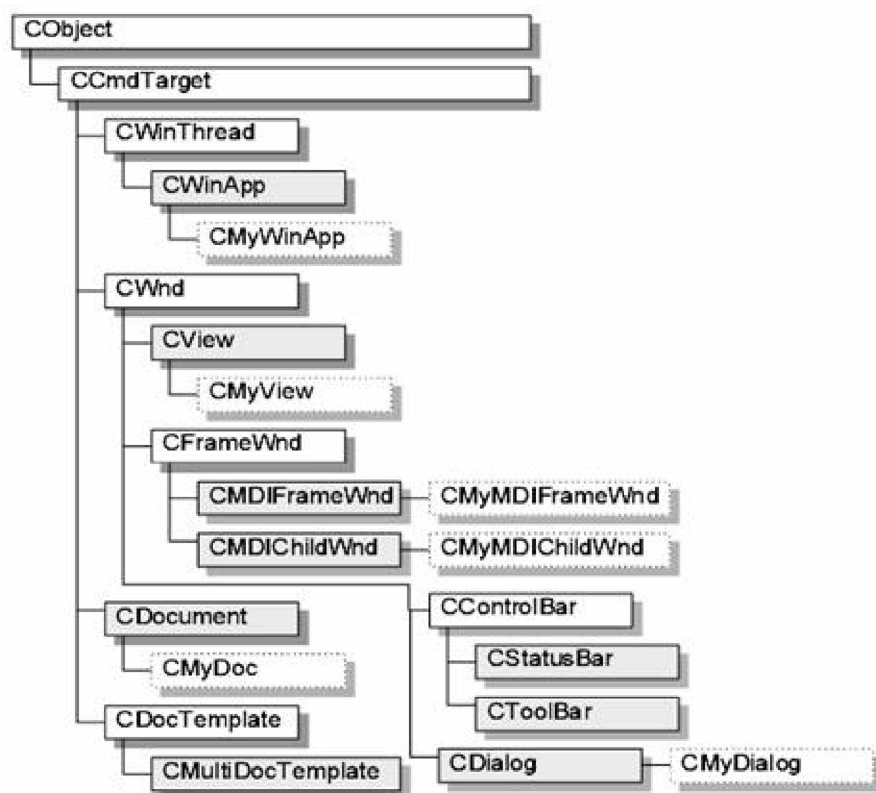


图7-1 本章范例程序所使用的MFC类。请与图6-1做比较。

MFC 程序的 UI 新风貌

一套好软件少不得一幅漂亮的用户接口。图7-2是信手拈来的几个知名 Windows 软件，它们一致具备了工具栏和状态栏等视觉对象，并拥有MDI风格。利用MFC，我们很轻易就能够做出同等级的UI接口。

撰写MFC程序，我们一定要放弃传统的「纯手工打造」方式，改用Visual C++ 提供的各种开发工具。AppWizard 可以为我们制作出 MFC 程序骨干；只要选择某些按钮，不费吹灰之力你就可以获得一个很漂亮的程序。这个全自动生产线做出来的程序虽不具备任何特殊功能（那正是我们程序员的任务），但已经拥有以下的特征：

标准的【File】菜单，以及对话框。

标准的【Edit】菜单（剪贴板功能）。这份菜单是否一开始就有功效，必须视你选用哪一种View而定，例如CEditView就内建有剪贴板功能。

标准MDI程序应该具备的【Window】菜单

【Help】菜单和 About 对话框亦已备妥。

此外，标准的工具栏和状态栏也已备妥，并与菜单内容建立起映射关系。所谓工具栏，是将某几个常用的菜单项目以按钮型式呈现出来，有一点热键的味道。这个工具栏可以随处停驻（dockable）。所谓状态栏，是主窗口最下方的文字显示区；只要菜单拉下，状态列就会显示鼠标座落的菜单项目的说明文字。状态栏右侧有三个小窗口（可扩充个数），用来显示一些特殊按键的状态。

打印与预览功能也已是半成品。【File】菜单拉下来可以看到【Print...】和【Print Preview】两项目：

骨干程序的Document和View目前都还是白纸一张，需要我们加工，所以一开始看不出打印与预览的真正功能。但如果我们在 AppWizard 中选用的View 类是 CEditView（如同第4 章），用户就可以打印其编辑成果，并可以在打印之前预览。也就是说，一进程序代码都不必写，我们就获得了一个可以同时编辑多份文件的文字编辑软件。

Document/View 支撑你的应用程序

我已经多次强调，Document/View 是 MFC 进化为 Application Framework 的灵魂。这个特征表现于程序设计技术上远多于表现在用户接口上，因此用户可能感觉不到什么是Document/View。程序员呢？程序员将因陌生而有一段阵痛期，然后开始享受它带来的便利。

我们在OLE中看到各对象（注）的集合称为一份Document；在MDI中看到子窗口所掌握的数据称为一个Document；现在在 MFC 又看到Document。"Document" 如今处处可见，再过不多久八成也要和"Object" 一样地泛滥了。

OLE对象指的是PaintBrush 完成的一张bitmap、SoundRecorder完成的一段Wave声音、Excel 完成的一份电子表格、Word 完成的一份文字等等等。为了恐怕与 C++ 的「对象」混淆，有些书籍将 OLE object 称为 OLE item。

在MFC之中，你可以把Document简单想作是「数据」。是的，只是数据，那么MFC的CDocument简单地说就是负责处理数据的类。

问题是，一个预先写好的类怎么可能管理未知的数据呢？MFC 设计之际那些伟大的天才们并不知道我们的数据结构，不是吗?! 他怎么知道我的程序要处理的数据是简单如：

```
char  name[20];
char  address[30];
int   age;
bool  sex;
```

或是复杂如：

```
struct dbllistnode
       {
        struct dbllistnode *next, *prev;
        struct info_t
              {
               int left;
               int top;

         int width;
         int height;
         void (*cursor)();
        } *item;
};
```

的确，预先处理未知的数据根本是不可能的。CDocument 只是把空壳做好，等君入瓮。它可以内嵌其它对象（用来处理基层数据类型如串列、数组等等），所以程序员可以在Document中拼拼凑凑出实际想要表达的文件完整格式。下一章进入 Scribble 程序的实际设计时，你就能够感受这一点。

CDocument 的另一价值在于它搭配了另一个重要的类：CView。

不论什么型式，数据总是有体有面。实际的数据数值就是体，显示在屏幕上（甚而印表机上）的画面就是面（图 7-3a）。「数值的处理」应该使用字节、整数、浮点数、串列、数组等数据结构，而「数值的表现」应该使用绘图工具如坐标系统、笔刷颜色、点线圆弧、字形...。CView就是为了数据的表现而设计的。

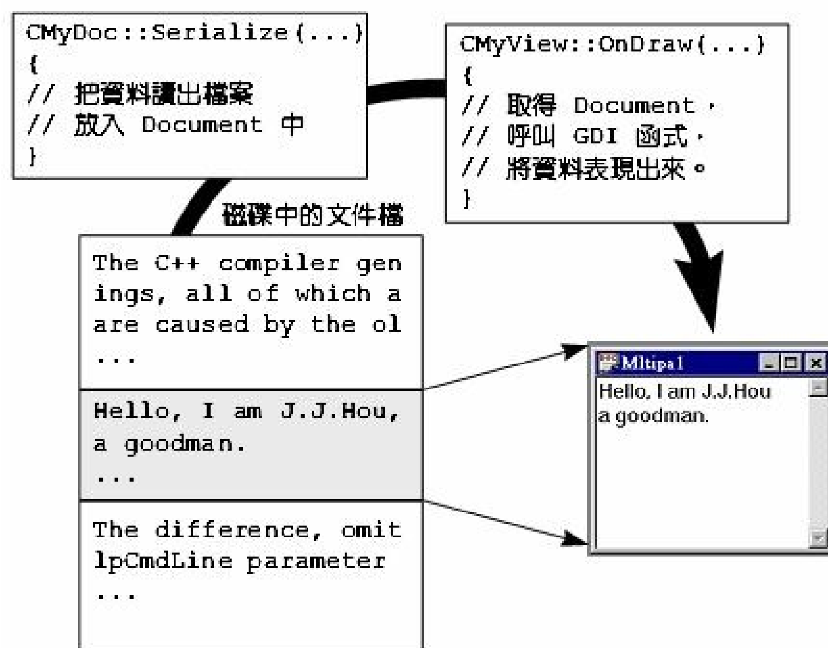


图7-3a Document是数据的体，View是数据的面。

除了负责显示，View 还负责程序与用户之间的交谈接口。用户对数据的编辑、修改都需仰赖窗口上的鼠标与键盘动作才得完成，这些消息都将由View接受后再通知Document(图7-3b)。

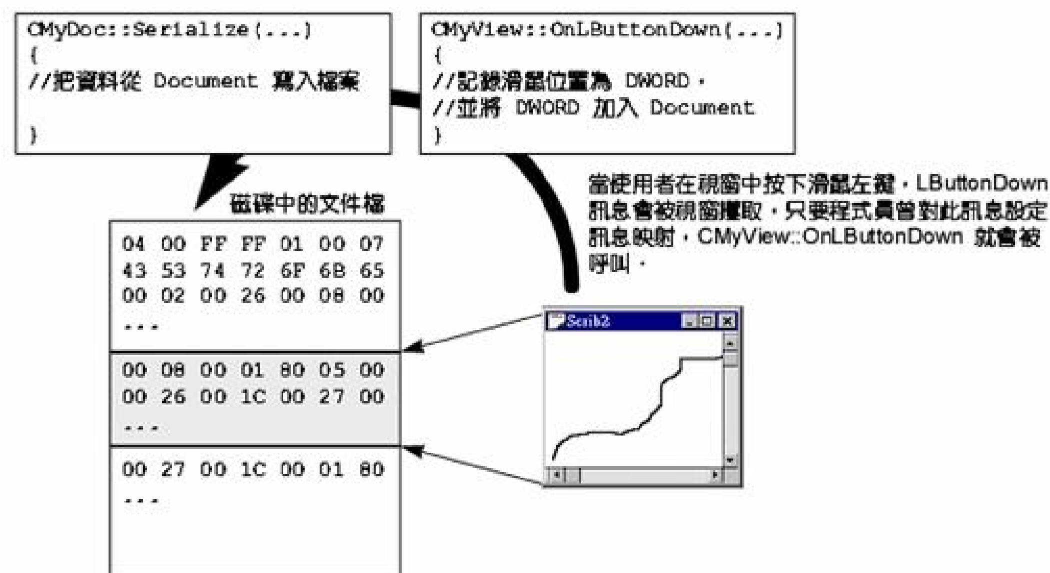


图7-3b View 是Document的第一线，负责与用户接触。

Document/View 的价值在于，这些 MFC 类已经把一个应用程序所需的「数据处理与显示」的函数空壳都设计好了，这些函数都是虚函数，所以你可以（也应该）在派生类中改写它们。有关文件读写的动作在 CDocument 的 Serialize 函数进行，有关画面显示的动作在 CView 的 OnDraw 或 OnPaint 函数进行。当我为自己派生两个类 CMyDoc和 CMyView，我只要把全付心思花在 CMyDoc::Serialize 和 CMyView::OnDraw 身上，其它琐事一概不必管，整个程序自动会运作得好好的。

什么叫做「整个程序会自动运作良好」？以下是三个例子：

- 如果按下【File/Open】，Application Framework 会启动对话框让你指定文件名，然后自动调用CMyDoc::Serialize 读文件 。Application Framework还会调用CMyView::OnDraw，把数据显示出来。
- 如果萤幕状态改变，产生了WM_PAINT，Framework会自动调用你的CMyView::OnDraw，传一个Display DC 让你重新绘制窗口内容。
- 如果按下【File/Print...】，Framework会自动调用你的CMyView::OnDraw，这次传进去的是个 Printer DC，因此绘图动作的输出对象就成了打印机。

MFC 已经把程序大架构完成了，模块与模块间的消息流动路径以及各函数的功能职司都已确定好（这是MFC之所以够格称为一个Framework 的原因），所以我们写程序的焦点就放在那些必须改写的虚函数身上即可。软件界当初发展 GUI 系统时，目的也是希望把程序员的心力导引到应用软件的真正目标去，而不必花在用户接口上。MFC 的Document/View 架构希望更把程序员的心力导引到真正的数据结构设计以及真正的数据显示动作上，而不要花在模块的沟通或消息的流动传递上。今天，程序员都对GUI称便，Document/View 也即将广泛地证明它的贡献。

Application Framework 使我们的程序写作犹如做填充题；Visual C++的软件开发工具则使我们的程序写作犹如做选择题。我们先做选择题，再在骨干程序中做填充题。的确，程序员的生活愈来愈像侯捷所言「只是软件IC装配厂里的男工女工」了。

现在让我们展开 MFC 深度之旅，彻底把 MFC 骨干程序的每一行都搞清楚。你应该已经从上一章具体了解了 MFC 程序从启动到结束的生命过程，这一章的例子虽然比较复杂，程序的生命过程是一样的。我们看看新添了什么内容，以及它们如何运作。我将以AppWizard 完成的Scribble Step0（第 4 章）为解说对象，一行不改。然后我会做一点点修改，使它成为一个多窗口文字编辑器。

利用 Visual C++ 工具完成 Scribble step0

我已经在第 4 章示范过 AppWizard 的使用方法，并实际制作出 Scribble Step0 程序，这里就不再重复说明了。完整的骨干程序原始代码亦已列于第 4 章。

这些由「生产线」做出来的程序代码其实对初学者并不十分合适，原因之一是容易眼花撩乱，有许多 #if...#endif、批注、奇奇怪怪的符号（例如 //{ 和 //}）；原因之二是每一个类有自己的 .H 文件和 .CPP 文件，整个程序因而幅员辽阔（六个 .CPP 文件和六个 .H 文件）。

图7-4是Scribble step0程序中各类的相关数据。

類別名稱	基礎類別	類別宣告於	類別定義於
CScribbleApp	*CWinApp*	Scribble.h	Scribble.cpp
CMainFrame	*CMDIFrameWnd*	Mainfrm.h	Mainfrm.cpp
CChildFrame	*CMDIChildWnd*	Childfrm.h	Childfrm.cpp
CScribbleDoc	*CDocument*	ScribbleDoc.h	ScribbleDoc.cpp
CScribbleView	*CView*	ScribbleView.h	ScribbleView.cpp
CAboutDlg	*CDialog*	Scribble.cpp	Scribble.cpp

图7-4 Scribble 骨干程序中的重要组成份子

骨干程序使用哪些 MFC 类？

对，你看到的Scribble step0就是一个完整的MFC应用程序，而我保证你一定昏头转向茫无头绪。没有关系，我们才刚启航。

如果把标准图形接口（工具栏和状态栏）以及 Document/View 考虑在内，一个标准的MFC MDI程序使用这些类：

MFC 類別名稱	我的類別名稱	功能
CWinApp	*CScribbleApp*	application object
CMDIFrameWnd	*CMainFrame*	MDI 主視窗
CMultiDocTemplate	直接使用	管理 Document/View
CDocument	*CScribbleDoc*	Document，負責資料結構與檔案動作
CView	*CScribbleView*	View，負責資料的顯示與印表
CMDIChildWnd	*CChildFrame*	MDI 子視窗
CToolBar	直接使用	工具列
CStatusBar	直接使用	狀態列
CDialog	*CAboutDlg*	About 對話盒

应用程序各显身手的地方只是各个可被改写的虚函数。这九个类在MFC的地位请看图7-1。下一节开始我会逐项解释每一个对象的产生时机及其重要性质。

Document/View不只应用在MDI程序，也应用在SDI程序上。你可以在 AppWizard 的「Options 对话框」（图 4-2b）选择SDI风格。本书以MDI程序为讨论对象。

为了对标准的MFC程序有一个大局观，图7-4显示Scribble step0中各重要组成份子（类），这些组成份子在运行时期的意义与主从关系显示于图 7-5。

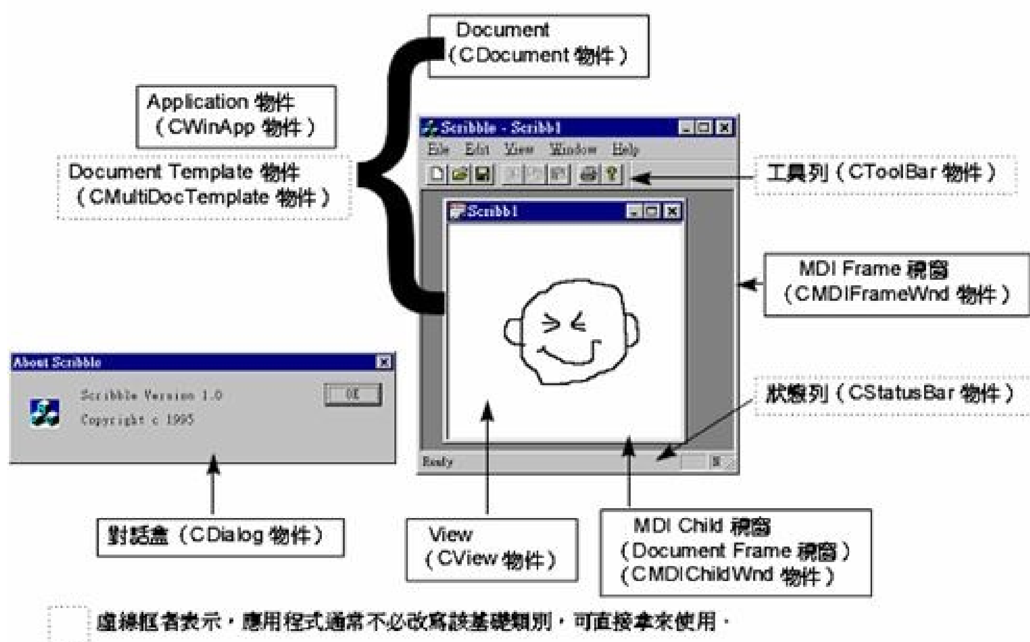


图7-5 Scribble step0程序中的九个对象（几乎每个MFC MDI程序都如此）

图7-6是 Scribble step0程序缩影，我把运行时序标上去，对于整体概念的形成将有帮助。

```
#0001 class CScribbleApp : public CWinApp
#0002 {
#0003     virtual BOOL InitInstance();    // 注意：第6章的 HelloMFC 程式是在 CFrameWnd
#0004     afx_msg void OnAppAbout();      // 類別中處理 "About" 命令，這裡的 Scribble
#0005     DECLARE_MESSAGE_MAP()           // 程式卻在 CWinApp 衍生類別中處理之。到底，
#0006 };                                  // 一個訊息可以（或應該）在哪裡被處理才是合理？
#0007                                     // 第 9 章「訊息映射與命令繞行」可以解決這個疑惑。
#0008 class CMainFrame : public CMDIFrameWnd
#0009 {
#0010     DECLARE_DYNAMIC(CMainFrame)
#0011     CStatusBar  m_wndStatusBar;
#0012     CToolBar    m_wndToolBar;
#0013     afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
#0014     DECLARE_MESSAGE_MAP()
#0015 };
#0016
#0017 class CChildFrame : public CMDIChildWnd
#0018 {
#0019     DECLARE_DYNCREATE(CChildFrame)
#0020     DECLARE_MESSAGE_MAP()
#0021 };
#0022
#0023 class CScribbleDoc : public CDocument
#0024 {
#0025     DECLARE_DYNCREATE(CScribbleDoc)
#0026     virtual void Serialize(CArchive& ar);
#0027     DECLARE_MESSAGE_MAP()
#0028 };
#0029
#0030 class CScribbleView : public CView
```

```
#0031 {
#0032     DECLARE_DYNCREATE(CScribbleView)
#0033     CScribbleDoc* GetDocument();
#0034
#0035     virtual void OnDraw(CDC* pDC);
#0036    DECLARE_MESSAGE_MAP()
#0037 };
#0038
#0039 class CAboutDlg : public CDialog
#0040 {
#0041         DECLARE_MESSAGE_MAP()
#0042 };
#0043
#0044 //-----------------------------------
#0045
#0046 CScribbleApp theApp;  ①
```

```
// MFC 内部
Int AfxAPI AfxWinMain(¡K
{
  CWinApp* pApp = AfxGetApp();
  AfxWinInit(¡K);                ②
  pApp->InitApplication();       ③
  pApp->InitInstance();          ④
  nReturnCode = pApp->Run();     ❶
```

```
#0047
#0048 BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
#0049     ON_COMMAND(ID_APP_ABOUT, OnAppAbout)  ❹
#0050     ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
#0051     ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)  ❷
#0052     ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
#0053 END_MESSAGE_MAP()
#0054
#0055 BOOL CScribbleApp::InitInstance()  ⑤
#0056 {
#0057     ...
#0058     CMultiDocTemplate* pDocTemplate;
#0059     pDocTemplate = new CMultiDocTemplate(  ⑥
#0060             IDR_SCRIBTYPE,
#0061             RUNTIME_CLASS(CScribbleDoc),
#0062             RUNTIME_CLASS(CChildFrame),
#0063             RUNTIME_CLASS(CScribbleView));
#0064     AddDocTemplate(pDocTemplate);

#0065
#0066     CMainFrame* pMainFrame = new CMainFrame;  ⑦
#0067     pMainFrame->LoadFrame(IDR_MAINFRAME);      ⑧
#0068     m_pMainWnd = pMainFrame;
#0069     ¡K
#0070     pMainFrame->ShowWindow(m_nCmdShow);  ⑩
#0071     pMainFrame->UpdateWindow();
#0072     return TRUE;
#0073 }
#0074
#0075 BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
#0076 END_MESSAGE_MAP()
#0077
#0078 void CScribbleApp::OnAppAbout()  ❺
#0079 {
#0080     CAboutDlg aboutDlg;
#0081     aboutDlg.DoModal();
#0082 }
#0083
#0084 IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)
#0085
#0086 BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
#0087     ON_WM_CREATE()
#0088 END_MESSAGE_MAP()
#0089
#0090 static UINT indicators[] =
#0091 {
#0092     ID_SEPARATOR,
#0093     ID_INDICATOR_CAPS,
#0094     ID_INDICATOR_NUM,
#0095     ID_INDICATOR_SCRL,
#0096 };
```

```
// MFC 内部
CFrameWnd::Create
    ↓
CWnd::CreateEx
    ↓
::CreateWindowEx
```

WM_CREATE

```
#0097
#0098 int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)  ⑨
#0099 {
#0100     m_wndToolBar.Create(this);
#0101     m_wndToolBar.LoadToolBar(IDR_MAINFRAME);
#0102     m_wndStatusBar.Create(this);
#0103     m_wndStatusBar.SetIndicators(indicators,
#0104              sizeof(indicators)/sizeof(UINT));
#0105
#0106     m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
#0107           CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
#0108
#0109     m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
#0110     EnableDocking(CBRS_ALIGN_ANY);
#0111     DockControlBar(&m_wndToolBar);
#0112     return 0;
#0113 }
#0114
#0115 IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)
#0116
#0117 BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
#0118 END_MESSAGE_MAP()
#0119
#0120 IMPLEMENT_DYNCREATE(CScribbleDoc, CDocument)
#0121
#0122 BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
#0123 END_MESSAGE_MAP()
#0124
#0125 void CScribbleDoc::Serialize(CArchive& ar)  ❸
#0126 {
#0127     if (ar.IsStoring())
#0128     {
#0129            // TODO: add storing code here
#0130     }
#0131     else
#0132     {
#0133            // TODO: add loading code here
#0134     }
#0135 }
#0136
#0137 IMPLEMENT_DYNCREATE(CScribbleView, CView)
#0138
#0139 BEGIN_MESSAGE_MAP(CScribbleView, CView)
#0140     ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
#0141     ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
#0142     ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
#0143 END_MESSAGE_MAP()
#0144
#0145 void CScribbleView::OnDraw(CDC* pDC)
#0146 {
#0147     CScribbleDoc* pDoc = GetDocument();
#0148     // TODO: add draw code for native data here
#0149 }
```

CAP NUM SCRL

图7-6 Scribble step0 运行时序。这是一张简图，

有一些次要动作（例如滑鼠拉曳功能、设定对话框底色）并未列出，

但是在稍后的细部讨论中会提到。

以下是图 7-6 程序流程之说明：

①~④动作与流程和前一章的Hello程序如出一辙。

⑤我们改写InitInstance这个虚函数。

⑥ new 一个CMultiDocTemplate对象，此对象规划Document、View 以及 Document Frame 窗口三者之关系。

⑦new一个CMyMDIFrameWnd 对象，做为主窗口对象。

⑧调用LoadFrame，产生主窗口并加挂菜单等诸元，并指定窗口标题、文件标题、文件文件扩展名等（关键在 IDR_MAINFRAME 常数）。LoadFrame 内部将调用 Create，后者将调用 CreateWindowEx，于是触发WM_CREATE 消息。

⑨由于我们曾于CMainFrame之中拦截WM_CREATE（利用 ON_WM_CREATE 宏），所以 WM_CREATE 产生之际Framework会调用OnCreate。我们在此为主窗口挂上工具列和状态栏。

⑩回到InitInstance，执行ShowWindow 显示窗口。

a.InitInstance 结束，回到AfxWinMain，执行Run，进入消息循环。其间的黑盒子已在上一章的 Hello 范例中挖掘过。

b.消息经由Message Routing 机制，在各类的Message Map中寻求其处理例程。WM_COMMAND/ID_FILE_OPEN 消息将由 CWinApp::OnFileOpen 函数处理。此函数由 MFC 提供，它在显示过【File Open】对话框后调用 Serialize 函数。

c.我们改写Serialize函数以进行我们自己的文件读写动作。

d.WM_COMMAND/ID_APP_ABOUT 消息将由 OnAppAbout 函数处理。

e.OnAppAbout 函数利用 CDialog 的性质很方便地产生一个对话框。

Document Template 的意义

Document Template 是一个全新的观念。

稍早我已提过 Document/View 的概念，它们互为表里。View 本身虽然已经是一个窗口，其外围却必须再包装一个外框窗口做为舞台。这样的切割其实是为了让 View 可以非常独立地放置于「MDI Document Frame 窗口」或「SDI Document Frame 窗口」或「OLE Document

Frame 窗口」等各种应用之中。也可以说，Document Frame 窗口是 View 窗口的一个容器。数据的内容、数据的表象、以及「容纳数据表象之外框窗口」三者是一体的，换言之，程序每打开一份文件（数据），就应该产生三份对象：

1. 一份Document 对象，

2. 一份View 对象，

3. 一份CMDIChildWnd 对象（做为外框窗口）

这三份对象由一个所谓的Document Template对象来管理。让这三份对象产生关系的关键在于CMultiDocTemplate：

```
BOOL CScribbleApp::InitInstance()
{
    ...
    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
            IDR_SCRIBTYPE,
            RUNTIME_CLASS(CScribbleDoc),
            RUNTIME_CLASS(CChildFrame),
            RUNTIME_CLASS(CScribbleView));
    AddDocTemplate(pDocTemplate);
    ...
}
```

如果程序支持不同的数据格式（例如一为TEXT一为BITMAP），那么就需要不同的Document Template：

```
BOOL CMyWinApp::InitInstance()
{
  ...
  CMultiDocTemplate* pDocTemplate;

  pDocTemplate = new CMultiDocTemplate(
           IDR_TEXTTYPE,
           RUNTIME_CLASS(CTextDoc),
           RUNTIME_CLASS(CChildFrame),
           RUNTIME_CLASS(CTextView));
  AddDocTemplate(pDocTemplate);

  pDocTemplate = new CMultiDocTemplate(
           IDR_BMPTYPE,
           RUNTIME_CLASS(CBmpDoc),
           RUNTIME_CLASS(CChildFrame),
           RUNTIME_CLASS(CBmpView));
  AddDocTemplate(pDocTemplate);
  ...
}
```

这其中有许多值得讨论的地方，而 CMultiDocTemplate 的构造函数参数透露了一些端倪：

```
CMultiDocTemplate::CMultiDocTemplate(UINT nIDResource,
                                     CRuntimeClass* pDocClass,
                                     CRuntimeClass* pFrameClass,
                                     CRuntimeClass* pViewClass);
```

1.nIDResource：这是一个资源ID，表示此一文件类型（文件格式）所使用的资源。本例为IDR_SCRIBTYPE，在RC文件中代表多种资源（不同种类的资源可使用相同的ID）：

```
IDR_SCRIBTYPE ICON DISCARDABLE  "res\\ScribbleDoc.ico"
IDR_SCRIBTYPE MENU PRELOAD DISCARDABLE
{ ... }
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
   IDR_MAINFRAME "Scribble Step0"
   IDR_SCRIBTYPE "\nScrib\nScrib\nScribble Files (*.scb)\n.SCB\n
                  Scribble.Document\nScrib Document"
END
```

File Edit View Window Help

原碼太長，我把它截爲兩半，實際上是一整行。有七個子字串各以 '\n' 分隔。這些子字串完整描述文件的型態。第一個子字串於 MDI 程式中用不著，故本例省略（成爲空字串）。

其中的ICON是文件窗口被最小化之后的图示；MENU是当程序存在有任何文件窗口时所使用的菜单（如果没有开启任何文件窗口，菜单将是另外一套，稍后再述）。至于字符串表格（STRINGTABLE）中的字符串，稍后我有更进一步的说明。

2.pDocClass。这是一个指标，指向Document类别（衍生自 CDocument）之「CRuntimeClass 对象」。

3.pFrameClass。这是一个指针，指向Child Frame类（派生自 CMDIChildWnd）之「CRuntimeClass 对象」。

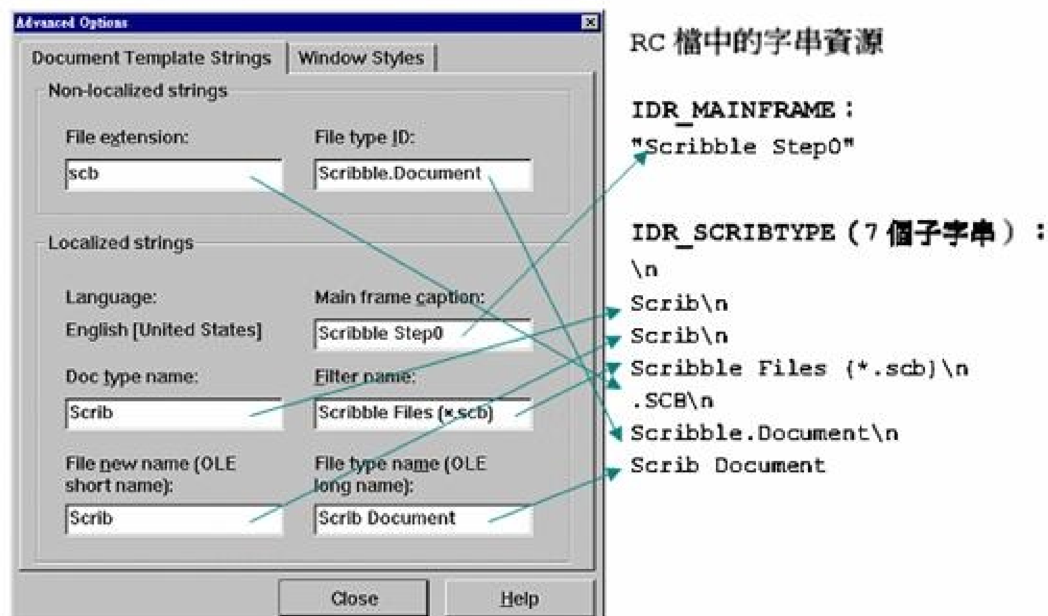
4.pViewClass。这是一个指针，指向View 类（派生自CView）之「CRuntimeClass对象」。

CRuntimeClass

我曾经在第3章「自制 RTTI」一节解释过什么是 CRuntimeClass。它就是「类型录网」串列中的元素类型。任何一个类只要在声明时使用 DECLARE_DYNAMIC或DECLARE_DYNCREATE或DECLARE_SERIAL宏，就会拥有一个静态的（static）CRuntimeClass 内嵌对象。

好，你看，Document Template 接受了三种类的CRuntimeClass 指针，于是每当用户打开一份文件，Document Template 就能够根据「类型录网」（第3章所述），动态生成出三个对象（document、view、document frame window）。如果你不记得 MFC 的动态生成是怎么一回事儿，现在正是复习第3章的时候。我将在第8章带你实际看看Document Template 的内部动作。

前面曾提到，我们在CMultiDocTemplate构造函数的第一个参数置入 IDR_SCRIBTYPE，代表RC文件中的菜单（MENU）、图标（ICON）、字符串（STRING）三种资源，其中又以字符串资源大有学问。这个字符串以 '\n' 分隔为七个子字符串，用以完整描述文件类型。七个子字符串可以在 AppWizard 的步骤四的【Advanced Options】对话框中指定：



每一个子字符串都可以在程序进行过程中取得，只要调用 CDocTemplate::GetDocString 并在其第二参数中指定索引值（1~7）即可，但最好是以 CDocTemplate 所定义的七个常数代替没有字面意义的索引值。下面就是 CDocTemplate 的 7 个常数定义：

```
// in AFXWIN.H
class CDocTemplate : public CCmdTarget
{
  ...
  enum DocStringIndex
  {
    windowTitle,  // default window title
    docName,      // user visible name for default document
    fileNewName,  // user visible name for FileNew

    // for file based documents:
    filterName,   // user visible name for FileOpen
    filterExt,    // user visible extension for FileOpen

    // for file based documents with Shell open support:
    regFileTypeId,    // REGEDIT visible registered file type identifier
    regFileTypeName   // Shell visible registered file type name
  };
  ...
};
```

所以，你可以这么做：

```
CString strDefExt, strDocName;
pDocTemplate->GetDocString(strDefExt, CDocTemplate::filterExt);
pDocTemplate->GetDocString(strDocName, CDocTemplate::docName);
```

七个子字符串意义如下：

Index	意义
1.CDocTemplate::windowTitle	主窗口标题栏上的字符串。SDI程序才需要指定它，MDI程序不需要指定，将以IDR_MAINFRAME字符串为默认值。
2. CDocTemplate::docName	文件基底名称（本例为"Scrib"）。这个名称再加上一个流水号代码，即成为新文件的名称（例如"Scrib1"）。如果此字符串未被指定，文件预设名称为 "Untitled"。
3. CDocTemplate::fileNewName	文件类型名称。如果一个程序支持多种文件，此字符串将显示在【File/New】对话框中。如果没有指明，就不能够在【File/New】对话框中处理此种文件。本例只支持一种文件类型，所以当你选按【File/New】，并不会出现对话框。第 13 章将示范「一个程序支持多种文件」的作法。
4. CDocTemplate::filterName	文件类型以及一个适用于此类型之万用过滤字符串(wildcard filter string)。本例为 "Scribble(*.scb)"。这个字符串将出现在【File Open】对话框中的【List Files Of Type】列示盒中。
5. CDocTemplate::filterExt	文件文件之扩展名，例如 "scb"。如果没有指明，就不能够在【File Open】对话框中处理此种文件文件。
6. CDocTemplate::regFileTypeId	如果你以 ::RegisterShellFileTypes 对系统的登录数据库（Registry）注册文件类型，此值会出现在HKEY_CLASSES_ROOT 之下成为其子机代码（subkey）并仅供 Windows 内部使用。如果未指定，此种文件类型就无法注册，鼠标拖放 (drag and drop) 功能就会受影响。
7. CDocTemplate::regFileTypeName	这也是储存在登录数据库（Registry）中的文件类型名称，并且是给人（而非只给系统）看的。它也会显示于程序中用以处理登录数据库之对话框内。

以下是Scribble范例中各个字符串出现的位置：

我必须再强调一次，AppWizard 早已帮我们产生出这些字符串。把这些来龙去脉弄清楚，只是为了知其所以然。当然，同时也为了万一你不喜欢AppWizard 准备的字符串内容，你知道如何去改变它。

Scribble 的 Document/View 设计

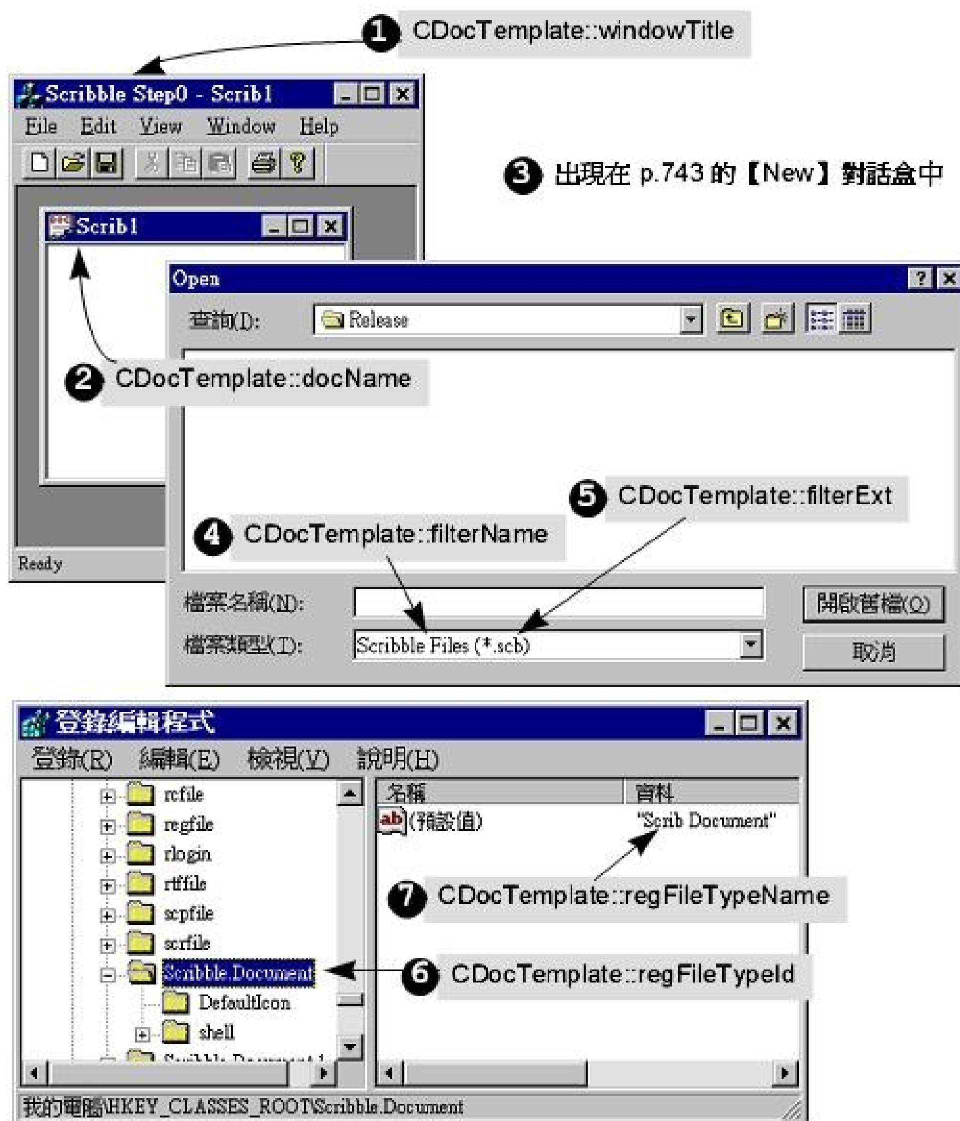
用最简单的一句话描述，Document 就是数据的体，View 就是数据的面。我们藉CDocument 管理数据，藉Collections Classes（MFC中的一组专门用来处理数据的类）处理实际的数据数据；我们藉CView 负责数据的显示，藉CDC和 CGdiObject 实际绘图。人们常说一体两面一体两面，在MFC中一体可以多面：同一份数据可以文字描述之，可以长条图描述之，亦可以曲线图描述之。

Document/View 之间的关系可以图7-3 说明。View 就像一个观景器（我避免使用「窗口」这个字眼，以免引起不必要的联想），用户透过 View 看到 Document，也透过 View 改变 Document。View 是 Document 的外显接口，但它并不能完全独立，它必须依存在一个所谓的 Document Frame 窗口内。

一份Document可以映射给许多个Views 显示，不同的Views可以对映到同一份巨大Document的不同局部。总之，请把View想象是一个镜头，可以观看大画布上的任何局部（我们可以选用 CScrollView 使之具备滚动条）；在镜头上加特殊的偏光镜、柔光镜、十字镜，我们就会看到不同的影像 -- 虽然观察的对象完全相同。

数据的管理动作有哪些？读文件和写文件都是必要的，文件存取动作称为 Serialization，由 Serialize 函数负责。我们可以（而且也应该）在 CMyDoc 中改写Serialize 函数，使它符合个人需求。数据格式的建立以及文件读写功能将在Scribble step1 中加入，本例（step0）的 CScribbleDoc 中并没有什么成员变量（也就是说容纳不了什么数据），Serialize 则简直是个空函数：

```
void CScribbleDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}
```



这也是我老说骨干程序啥大事也没做的原因。

除了文件读写，数据的显示也是必要的动作，数据的接受编辑也是必要的动作。两者都由View负责。用户对Document 的任何编辑动作都必须透过 Document Frame窗口，消息随后传到 CView。我们来想想我们的 View 应该改写哪些函数？

1. 当Document Frame窗口收到WM_PAINT，窗口内的View的OnPaint函数会被调用，OnPaint又调用OnDraw。所以为了显示数据，我们必须改写OnDraw。至于OnPaint，主要是做「只输出到屏幕而不到打印机」的动作。有关打印机，我将在第 12 章提到。

2. 为了接受编辑动作，我们必须在View 类中接受鼠标或键盘消息并处理之。如果要接受鼠标左键，我们应该改写View 类中的三个虚函数：

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
```

上述两个动作在 Scribble step0 都看不到，因为它是个啥也没做的程序：

```
void CScribbleView::OnDraw(CDC* pDC)
{
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
}
```

主窗口的诞生

上一章那个极为简单的 Hello 程序，主窗口采用CFrameWnd 类。本例是 MDI 风格，将采用 CMDIFrameWnd 类。

构造 MDI 主窗口，有两个步骤。第一个步骤是 new 一个 CMDIFrameWnd 对象，第二个步骤是调用其 LoadFrame 函数。此函数内容如下：

```
// in WNDFRM.CPP
BOOL CFrameWnd::LoadFrame(UINT nIDResource, DWORD dwDefaultStyle,
                          CWnd* pParentWnd, CCreateContext* pContext)
{
    CString strFullString;
    if (strFullString.LoadString(nIDResource))
        AfxExtractSubString(m_strTitle, strFullString, 0);  // first sub-string

    if (!AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG))
        return FALSE;

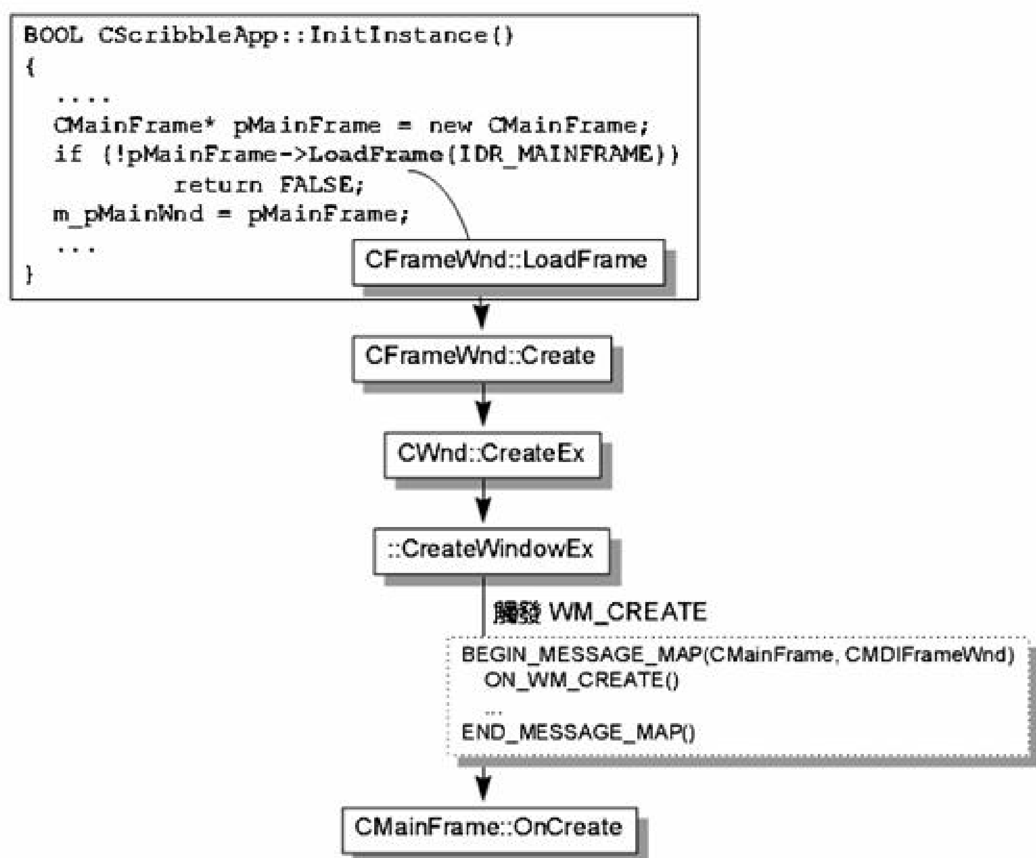
    // attempt to create the window
    LPCTSTR lpszClass = GetIconWndClass(dwDefaultStyle, nIDResource);
    LPCTSTR lpszTitle = m_strTitle;
    if (!Create(lpszClass, lpszTitle, dwDefaultStyle, rectDefault,
              pParentWnd, MAKEINTRESOURCE(nIDResource), 0L, pContext))
    {
          return FALSE;   // will self destruct on failure normally
    }

    // save the default menu handle
    m_hMenuDefault = ::GetMenu(m_hWnd);

    // load accelerator resource
    LoadAccelTable(MAKEINTRESOURCE(nIDResource));

    if (pContext == NULL)   // send initial update
          SendMessageToDescendants(WM_INITIALUPDATE, 0, 0, TRUE, TRUE);

    return TRUE;
}
```



窗口产生之际会发出WM_CREATE 消息，因此CMainFrame::OnCreate会被执行起来，那里将进行工具栏和状态栏的建立工作（稍后描述）。LoadFrame函数的参数（本例为IDR_MAINFRAME ）用来设定窗口所使用的各种资源，你可以从前一页的CFrameWnd::LoadFrame 原始代码中清楚看出。这些同名的资源包括：

```
// defined in SCRIBBLE.RC

IDR_MAINFRAME ICON DISCARDABLE  "res\\Scribble.ico"
IDR_MAINFRAME MENU PRELOAD DISCARDABLE  { ... }
IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE  { ... }
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
   IDR_MAINFRAME  "Scribble Step0"  （這個字串將成爲主視窗的標題）
   ..
END
```



这种作法（使用 LoadFrame 函数）与第 6 章的作法（使用Create函数）不相同，请注意。

工具栏和状态栏的诞生（Toolbar & Status bar）

工具栏和状态栏分别由 CToolBar 和 CStatusBar 掌管。两个对象隶属于主窗口，所以我们在CMainFrame 中以两个变量（事实上是两个对象）表示之：

```
class CMainFrame : public CMDIFrameWnd
{
protected:  // control bar embedded members
       CStatusBar  m_wndStatusBar;
       CToolBar    m_wndToolBar;
...
};
```

主窗口产生之际立刻会发出 WM_CREATE，我们应该利用这时机把工具栏和状态栏建立起来。为了拦截 WM_CREATE，首先需在 Message Map 中设定「映射项目」：

```
BEGIN_MESSAGE_MAP(CMyMDIFrameWnd, CMDIFrameWnd)
ON_WM_CREATE()
END_MESSAGE_MAP()
```

ON_WM_CREATE这个宏表示，只要WM_CREATE 发生，我的OnCreate函数就应该被调用。下面是由 AppWizard 产生的 OnCreate 标准动作：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
            return -1;

    if (!m_wndToolBar.Create(this) ||
            !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
            TRACE0("Failed to create toolbar\n");
            return -1;      // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
            !m_wndStatusBar.SetIndicators(indicators,
              sizeof(indicators)/sizeof(UINT)))
    {
            TRACE0("Failed to create status bar\n");
            return -1;      // fail to create
    }

 // TODO: Remove this if you don't want tool tips or a resizeable toolbar
 m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
       CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

 // TODO: Delete these three lines if you don't want the toolbar to
 //  be dockable
 m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
 EnableDocking(CBRS_ALIGN_ANY);
 DockControlBar(&m_wndToolBar);

 return 0;
```

其中有四个动作与工具栏和状态栏的产生及设定有关：

- m_wndToolBar.Create(this) 表示要产生一个隶属于this（也就是目前这个对象，也就是主窗口）的工具栏。

- m_wndToolBar.LoadToolBar(IDR_MAINFRAME) 将RC文件中 的工具列资源 载入。IDR_MAINFRAME 在RC文件中代表两种与工具栏有关的资源：

```
IDR_MAINFRAME BITMAP MOVEABLE PURE "RES\\TOOLBAR.BMP"
```



```
IDR_MAINFRAME TOOLBAR DISCARDABLE  16, 15
BEGIN
    BUTTON      ID_FILE_NEW
    BUTTON      ID_FILE_OPEN
    BUTTON      ID_FILE_SAVE
        SEPARATOR
    BUTTON      ID_EDIT_CUT
    BUTTON      ID_EDIT_COPY
    BUTTON      ID_EDIT_PASTE
        SEPARATOR
    BUTTON      ID_FILE_PRINT
    BUTTON      ID_APP_ABOUT
END
```

LoadToolBar函数一举取代了前一版的LoadBitmap+SetButtons 两个动 作 。LoadToolBar 知道如何把 BITMAP 资源和 TOOLBAR 资源搭配起来，完成工具栏的设定。当然啦，如果你不是使用 VC++资源工具来编辑工具栏，BITMAP 资源和TOOLBAR 资源就可能格数不符，那是不被允许的。TOOLBAR 资源中的各 ID 值就是菜单项目的子集合，因为所谓工具栏就是把比较常用的菜单项目集合起来以按钮方式提供给用户。

- m_wndStatusBar.Create(this) 表示要产生一个隶属于this对象（也就是目前这个对象，也就是主窗口）的状态栏。

- m_wndStatusBar.SetIndicators(,...) 的第一个参数是个数组；第二个参数是数组元素个数。所谓Indicator 是状态栏最右侧的「指示窗口」，用来表示大写键、数字键等的On/Off 状态。AFXRES.H 中定义有七种indicators：

```
#define ID_INDICATOR_EXT   0xE700  // extended selection indicator
#define ID_INDICATOR_CAPS  0xE701  // cap lock indicator
#define ID_INDICATOR_NUM   0xE702  // num lock indicator
#define ID_INDICATOR_SCRL  0xE703  // scroll lock indicator
#define ID_INDICATOR_OVR   0xE704  // overtype mode indicator
#define ID_INDICATOR_REC   0xE705  // record mode indicator
#define ID_INDICATOR_KANA  0xE706  // kana lock indicator
```

本例使用其中三种：

```
static UINT indicators[] =
{
        ID_SEPARATOR, // status line indicator
        ID_INDICATOR_CAPS,
        ID_INDICATOR_NUM,
        ID_INDICATOR_SCRL,
};
```

CAP NUM SCRL

鼠标拖放（Drag and Drop）

MFC 程序很容易拥有Drag and Drop功能。意思是，你可以从Shell（例如 Windows 95的文件总管）中以鼠标拉动一个文件，拖到你的程序中，你的程序因而打开此文件并读其内容，将内容放到一个Document Frame窗口中。甚至，用户在 Shell 中以鼠标对某个文件文件（你的应用程序的文件文件）快按两下，也能启动你这个程序，并自动完成开文件，读文件，显示等动作。

在SDK 程序中要做到Drag and Drop，并不算太难，这里简单提一下它的原理以及作法。当用户从 Shell 中拖放一个文件到程序 A，Shell 就配置一块全局内存，填入被拖曳的文件名称（包含路径），然后发出WM_DROPFILES传到程序 A 的消息队列。程序 A 取得此消息后，应该把内存的内容取出，再想办法开文件读文件。

并不是张三和李四都可以收到WM_DROPFILES，只有具备 WS_EX_ACCEPTFILES 风格的窗口才能收到此一消息。欲让窗口具备此一风格，必须使用 CreateWindowEx （而不是传统的 CreateWindow），并指定第一个参数为 WS_EX_ACCEPTFILES。

剩下的事情就简单了：想办法把内存中的文件名和其它信息取出（内存 handle 放在 WM_DROPFILES 消息的wParam 中）。这件事情有DragQueryFile 和 DragQueryPoint两个 API 函数可以帮助我们完成。

SDK 的方法真的不难，但是MFC程序更简单：

```
BOOL CScribbleApp::InitInstance()
{
    ...
    // Enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    // Enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);
    ...
}
```

这三个函数的用途如下：

- CWnd::DragAcceptFile(BOOL bAccept=TRUE); 参数TRUE表示你的主窗口以及每一个子窗口（文件窗口）都愿意接受来自Shell的拖放文件。CFrameWnd内有一个OnDropFiles成员函数，负责对WM_DROPFIELS消息做出反应，它会通知application对

象的OnOpenDocument（此函数将在第８章介绍），并夹带被拖放的文件的名称。

- CWinApp::EnableShellOpen(); 当用户在Shell中对着本程序的文件文件快按两下时，本程序能够打开文件并读内容。如果当时本程序已执行， Framework不会再执行起程序的另一副本，而只是以DDE（Dynamic Data Exchange，动态数据交换）通知程序把文件（文件）读进来。DDE 处理例程内建在CDocManager之中（第８章会谈到这个类）。也由于DDE的能力，你才能够很方便地把文件图标拖放到打印机图标上，将文件打印出来。

  通常此函数后面跟随着RegisterShellFileTypes。

- CWinApp::RegisterShellFileTypes(); 此函数将向Shell注册本程序的文件类型。有了这样的注册动作，用户在Shell的双击动作才有着力点。这个函数搜寻Document Template串列中的每一种文件类型，然后把它加到系统所维护的registry（登录数据库）中。

  在传统的Windows 程序中，对Registry 的注册动作不外乎两种作法，一是准备一个.reg文件，由用户利用 Windows 提供的一个小工具regedit.exe，将.reg合并到系统的Registry中。第二种方法是利用::RegCreateKey、::RegSetValue等Win32 函数， 直接编辑Registry。 MFC 程序的作法最简单，只要调用CWinApp::RegisterShellFileTypes 即可。

必须注意的是，如果某一种文件类型已经有其对应的应用程序（例如 .txt 对应Notepad，.bmp 对应PBrush，.ppt 对应PowerPoint，.xls 对应Excel），那么你的程序就不能够横刀夺爱。如果本例Scribble的文件文件扩展名为 .txt，用户在Shell中双击这种文件，启动的将是Notepad 而不是Scribble。

另一个要注意的是，拖放动作可以把任何类型的文件文件拉到你的窗口中，并不只限于你所注册的文件类型。你可以把.bmp文件从Shell拉到Scribble窗口，Scribble 程序一样会读它并为它准备一个窗口。想当然耳，那会是个无言的结局：



消息映射（Message Map）

每一个派生自 CCmdTarget 的类都可以有自己的Message Map以处理消息。首先你应该在类声明处加上DECLARE_MESSAGE_MAP 宏，然后在 .CPP文件中使用 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 两个宏，宏中间夹带的就是「消息与函数对映关系」的一笔笔记录。

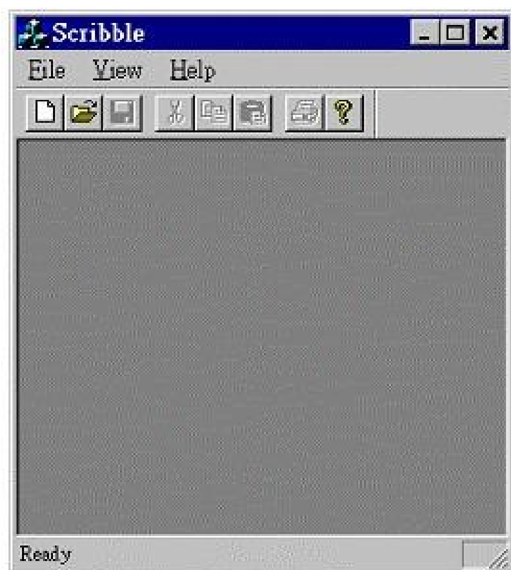
你可以从图 7-6 那个浓缩的Scribble原始代码中看到各类的Message Map。本例CScribbleApp类接受四个WM_COMMAND 消息：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

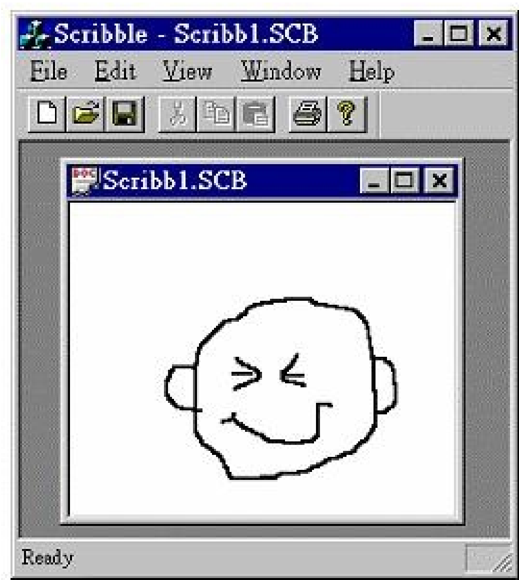
除了ID_APP_ABOUT是由我们自己设计一个OnAppAbout 函数处理之，其它三个消息都交给CWinApp成员函数去处理，因为那些动作十分制式，没什么好改写的。到底有哪些制式动作呢？看下一节！

标准菜单 File / Edit / View / Window / Help

仔细观察你所能搜集到的各种 MDI 程序，你会发现它们几乎都有两组菜单。一组是当没有任何子窗口（文件窗口）存在时出现（本例代码是 IDR_MAINFRAME）：



另一组则是当有任何子窗口（文件窗口）存在时出现（本例代码是 IDR_SCRIBTYPE）：



前者多半只有【File】、【View】、【Help】等选项，后者就复杂了，程序所有的功能都在上面。本例的IDR_MAINFRAME和IDR_SCRIBTYPE 就代表RC文件中的两组菜单。当用户打开一份文件文件，程序应该把主窗口上的菜单换掉，这个动作在SDK程序中由程序员负责，在MFC程序中则由Framework代劳了。

拉下这些菜单仔细瞧瞧，你会发现 Framework 真的已经为我们做了不少琐事。凡是菜单项目会引起对话框的，像是 Open 对话框、Save As 对话框、Print 对话框、Print Setup 对话框、Find 对话框、Replace 对话框，都已经恭候差遣；Edit 菜单上的每一项功能都已经可以应用在由 CEditView 掌控的文字编辑器上；File 菜单最下方记录着最近使用过的（所谓 LRU）四个文件名称（个数可在 Appwizard 中更改），以方便再开启；View 选单允许你把工具栏和状态栏设为可见或隐藏；Window 菜单提供重新排列子窗口图标的能力，以及对子窗口的排列管理，包括卡片式（Cascade）或拼贴式（Tile）。

下表是预设之菜单命令项及其处理例程的摘要整理。最后一个字段「是否预有关联」如果是Yes，意指只要你的程序菜单中有此命令项，当它被选按，自然就会引发命令处理例程，应用程序不需要在任何类的 Message Map 中拦截此命令消息。但如果是No，表示你必须在应用程序中拦截此消息。

菜单内容	命令项 ID	预设的处理函数	预有关联

File			
New	*ID_FILE_NEW*	*CWinApp::OnFileNew*	No
Open	*ID_FILE_OPEN*	*CWinApp::OnFileOpen*	No
Close	*ID_FILE_CLOSE*	*CDocument::OnFileClose*	Yes
Save	*ID_FILE_SAVE*	*CDocument::OnFileSave*	Yes
Save As	*ID_FILE_SAVEAS*	*CDocument::OnFileSaveAs*	Yes
Print	*ID_FILE_PRINT*	*CView::OnFilePrint*	No
Print Pre&view	*ID_FILE_PRINT_PREVIEW*	*CView::OnFilePrintPreview*	No
Print Setup	*ID_FILE_PRINT_SETUP*	*CWinApp::OnFilePrintSetup*	No
"Recent File Name"	*ID_FILE_MRU_FILE1~4*	*CWinApp::OnOpenRecentFile*	Yes
Exit	*ID_APP_EXIT*	*CWinApp::OnFileExit*	Yes
Edit			
Undo	*ID_EDIT_UNDO*	None	
Cut	*ID_EDIT_CUT*	None	
Copy	*ID_EDIT_COPY*	None	
Paste	*ID_EDIT_PASTE*	None	
View			
Toolbar	*ID_VIEW_TOOLBAR*	*FrameWnd::OnBarCheck*	Yes
Status Bar	*ID_VIEW_STATUS_BAR*	*FrameWnd::OnBarCheck*	Yes
Window（MDI only）			
New Window	*ID_WINDOW_NEW*	*MDIFrameWnd::OnWindowNew*	Yes
Cascade	*ID_WINDOW_CASCADE*	*MDIFrameWnd::OnWindowCmd*	Yes
Tile	*ID_WINDOW_TILE_HORZ*	*MDIFrameWnd::OnWindowCmd*	Yes
Arrange Icons	*ID_WINDOW_ARRANGE*	*MDIFrameWnd::OnWindowCmd*	Yes
Help			
About AppName	*ID_APP_ABOUT*	None	

上表的最后一字段为No者有五笔，表示虽然那些命令项有预设的处理例程，但你必须在自己的Message Map中设定映射项目，它们才会起作用。噢，AppWizard 此时又表现出了它的善体人意，自动为我们做出了这些代码：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
        ...
        ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
        ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
        ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CScribbleView, CView)
        ...
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

对话框

Scribble 可以启动许多对话框，前一节提了许多。唯一要程序员自己动手（我的意思是出现在我们的程序代码中）的只有 About 对话框。

为了拦截WM_COMMAND的ID_APP_ABOUT项目，首先我们必须设定其Message Map：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

当消息送来，就由OnAppAbout 处理：

```
void CScribbleApp::OnAppAbout()

{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}
```

其中CAboutDlg是CDialog的派生类：

```
class CAboutDlg : public CDialog
{
    enum { IDD = IDD_ABOUTBOX }; //IDD_ABOUTBOX是RC文件中的对话框面板资源
    ...
    DECLARE_MESSAGE_MAP()
};
```

比之于SDK 程序中的对话框，这真是方便太多了。传统SDK程序要在RC文件中定义对话框面板（dialog template，也就是其外形），在C程序中设计对话框函数。现在只需从CDialog派生出一个类，然后产生该类之对象，并指定 RC 文件中的对话框面板资源，再调用对话框对象的DoModal成员函数即可。

第10章一整章将讨论所谓的对话框数据交换（DDX）与对话框数据确认（DDV）。

改用 CEditView

Scribble step0 除了把一个应用程序的空壳做好，不能再贡献些什么。如果我们在AppWizard 步骤六中把 CScribbleView 的基类从 CView 改为 CEditView，那可就有大妙用了：

CEditView 是一个已具备文字编辑能力的类，它所使用的窗口是Windows 的标准控制组件之一 Edit，其SerializeRaw 成员函数可以把Edit控制组件中的raw text（而非「对象」所持有的数据）写到文件中。当我们在AppWizard 步骤六选择了它，程序代码中所有的CView统统变成CEditView，而最重要的两个虚函数则变成：

```
void CScribbleDoc::Serialize(CArchive& ar)
{
    // CEditView contains an edit control which handles all serialization
    ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
}

void CScribbleView::OnDraw(CDC* pDC)
{
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
}
```

就这样，我们不费吹灰之力获得了一个多窗口的文字编辑器，并拥有读写文件能力以及预览能力。

第四篇 深入MFC程序设计

第 8 章 Document-View 深入探讨

> 形而上者谓之道，形而下者谓之器。
>
> 对于 Document/View 而言，很少有人能够先道而后器。

完全由AppWziard代劳做出的Scribble step0，应用程序的整个架构（空壳）都已经构造起来了，但是Document和View还空着好几个最重要的函数（都是虚函数）等着你设计其实体。这就像一部汽车外面的车体以及内部的油路电路都装配好了，但还等着最重要的发动机（引擎）植入，才能够产生动力，开始「有所为」。

我已经在第 7 章概略介绍了Document/View以及Document Template，还有更多的秘密将在本章揭露。

为什么需要 Document-View（形而上）

MFC 之所以为 Application Framework，最重要的一个特征就是它能够将管理数据的程序代码和负责数据显示的程序代码分离开来，这种能力由MFC的 Document/View 提供。

Document/View是MFC的基石，了解它，对于有效运用MFC有极关键的影响。甚至OLE复合文件（compound document）都是建筑在Document/View的基础上呢！

几乎每一个软件都致力于数据的处理，毕竟信息以及数据的管理是计算机技术的主要用途。把数据管理和显示方法分离开来，需要考虑下列几个议题：

1. 程序的哪一部分拥有数据

2. 程序的哪一部分负责更新数据

3. 如何以多种方式显示数据

4. 如何让数据的更改有一致性

5. 如何储存数据（放到永久储存装置上）

6. 如何管理使用者接口。不同的数据类型可能需要不同的使用者接口，而一个程序可能管理多种类型的数据。

其实Document / View不是什么新主意，Xerox PARC实验室是这种观念的滥觞。它是 Smalltalk 环境中的关键性部分，在那里它被称为 Model-View-Controller（MVC）。其中的Model就是MFC的Document，而 Controller 相当于 MFC 的 Document Template。

回想在没有Application Framework帮助的时代（并不太久以前），你如何管理数据？只要程序需要，你就必须想出各种表现数据的方法；你有责任把数据的各种表现方法和资料本体调解出一种关系出来。100 位程序员，有 100 种作法！如果你的程序只处理一种数据类型，情况还不至于太糟。举个例，字处理软件可以使用巨大的字符串数组，把文字统统含括进来，并以 ASCII 型式显示之，顶多嘛，变换一下字形！

但如果你必须维护一种以上的数据类型，情况又当如何？想象得到，每一种数据类型可能需要独特的处理方式，于是需要一套功能选单；每一种数据类型显现在窗口中，应该有独特的窗口标题以及缩小图标；当数据编辑完毕要存盘，应该有独特的扩展名；登录在 Registry 之中应该有独特的型号。再者，如果你以不同的窗口，不同的显现方式，秀出一份数据，当数据在某一窗口中被编辑，你应该让每一窗口的数据显像与实际数据之间常保一致。吧啦吧啦吧啦……繁杂事务不胜枚举。

很快地，问题就浮显出来了。程序不仅要做数据管理，更要做「与数据类型相对应的 UI」的管理。幸运的是，解决之道亦已浮现，那就是面向对象观念中的Model-View-Controller（MVC），也就是MFC的Document/View。

Document

名称有点令人惧怕 -- Document 令我们想起文字处理软件或电子表格软件中所谓的「文件」。但，这里的Document其实就是数据。的确是，不必想得过份复杂。有人用data set 或 data source 来表示它的意义，都不错。

Document在MFC的CDocument里头被具体化。CDocument本身并无实务贡献，它只是提供一个空壳。当你开发自己的程序，应该从CDocument派生出一个属于自己的Document 类，并且在类中声明一些成员变量，用以承载（容纳）数据。然后再（至少）改写专门负责文件读写动作的Serialize函数。事实上，AppWizard为我们把空壳都准备好了，以下是Scribble step0 的部分内容：

```
class CScribbleDoc : public CDocument
{
        DECLARE_DYNCREATE(CScribbleDoc)
        ...
        virtual void Serialize(CArchive& ar);
        DECLARE_MESSAGE_MAP()
};
```

```
void CScribbleDoc::Serialize(CArchive& ar)
{
        if (ar.IsStoring())
        {
                // TODO: add storing code here
        }
        else
        {
                // TODO: add loading code here
        }
}
```

由于CDocument派生自CObject，所以它就有了CObject 所支持的一切性质，包括执行时期型别信息（RTTI）、动态生成（Dynamic Creation）、文件读写（Serialization）。又由于它也派生自CCmdTarget，所以它可以接收来自选单或工具列的WM_COMMAND 消息。

View

View 负责描述（呈现）Document 中的数据。

View 在MFC的CView里头被具体化。CView本身亦无实务贡献，它只是提供一个空壳。当你开发自己的程序，应该从CView派生出一个属于自己的View类，并且在类中（至少）改写专门负责显示数据的OnDraw函数（针对屏幕）或OnPrint函数（针对打印机）。事实上，AppWizard 为我们把空壳都准备好了，以下是Scribble step0 的部分内容：

```
class CScribbleView : public CView
{
        DECLARE_DYNCREATE(CScribbleView)
        ...
        virtual void OnDraw(CDC* pDC);
        DECLARE_MESSAGE_MAP()
};
```

```
void CScribbleView::OnDraw(CDC* pDC)
{
        CScribbleDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        // TODO: add draw code for native data here
}
```

由于CView派生自CWnd，所以它可以接收一般Windows 消息（如WM_SIZE、 WM_PAINT 等等），又由于它也派生自CCmdTarget，所以它可以接收来自选单或工具列的 WM_COMMAND 消息。

在传统的C/SDK 程序中，当窗口函数收到WM_PAINT，我们（程序员）就调用BeginPaint，获得一个Device Context（DC），然后在这个 DC 上作画。这个 DC 代表萤幕装置。在 MFC 里头，一旦 WM_PAINT 发生，Framework 会自动调用 OnDraw 函数。

View 事实上是个没有边框的窗口。真正出现时，其外围还有一个有边框的窗口，我们称为 Frame 窗口。

Document Frame（View Frame）

如果你的程序管理两种不同类型的数据，譬如说一个是TEXT，一个是 BITMAP，作为一位体贴的程序设计者，我想你很愿意为你的使用者考虑多一些：你可能愿意在使用者操作TEXT数据时，换一套TEXT专属的使用者接口，在使用者操作 BITMAP 数据时，换一套 BITMAP 专属的使用者接口。这份工作正是由 Frame 窗口负责。

乍见这个观念，我想你会惊讶为什么UI的管理不由View直接负责，却要交给Frame窗口？你知道，有时候机能与机能之间要有点黏又不太黏才好，把UI管理机能隔离出来，可以降低彼此之间的依存性，也可以使机能重复使用于各种场合如 SDI、MDI、OLE in-place editing（即地编辑）之中。如此一来View的弹性也会大一些。

Document Template

MFC把Document/View/Frame 视为三位一体。可不是吗！每当使用者欲打开（或新增）一份文件，程序应该做出Document、View、Frame 各一份。这个「三口组」成为一个运作单元，由所谓的Document Template 掌管。MFC有一个 CDocTemplate 负责此事。它又有两个派生类，分别是 CMultiDocTemplate 和 CSingleDocTemplate。所以我在上一章说了，如果你的程序能够处理两种数据类型，你必须制造两个Document Template 出来，并使用 AddDocTemplate 函数将它们一一加入系统之中。这和程序是不是MDI并没有关系。如果你的程序支持多种数据类型，但却是个 SDI，那只不过表示你每次只能开启一份文件罢了。

但是，逐渐地，MDI这个字眼与它原来的意义有了一些出入（要知道，这个字眼早在SDK时代即有了）。因此，你可能会看到有些书籍这么说：『MDI 程序使用 CMultiDocTemplate，SDI 程序使用CSingleDocTemplate』，那并不是很精准。

CDocTemplate 是个抽象类，定义了一些用来处理「Document/View/Frame 三口组」的基础函数。

CDocTemplate 管理 CDocument / CView / CFrameWnd

好，我们说Document Template 管理「三口组」，谁又来管理Document Template 呢？答案是CWinApp。下面就是InitInstance 中应有的相关作为：

```
BOOL CScribbleApp::InitInstance()
{
        ...
        CMultiDocTemplate* pDocTemplate;
        pDocTemplate = new CMultiDocTemplate(
                IDR_SCRIBTYPE,
                RUNTIME_CLASS(CScribbleDoc),
                RUNTIME_CLASS(CChildFrame),
                RUNTIME_CLASS(CScribbleView));
        AddDocTemplate(pDocTemplate);
        ...
}
```

想一想文件是怎么开启的：使用者选按【File/New】或【File/Open】（前者开启一份空档，后者读文件放到文件中），然后在View窗口内展现出来。我们很容易误以为是CWinApp直接产生Document：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

其实才不，是Document Template的杰作：

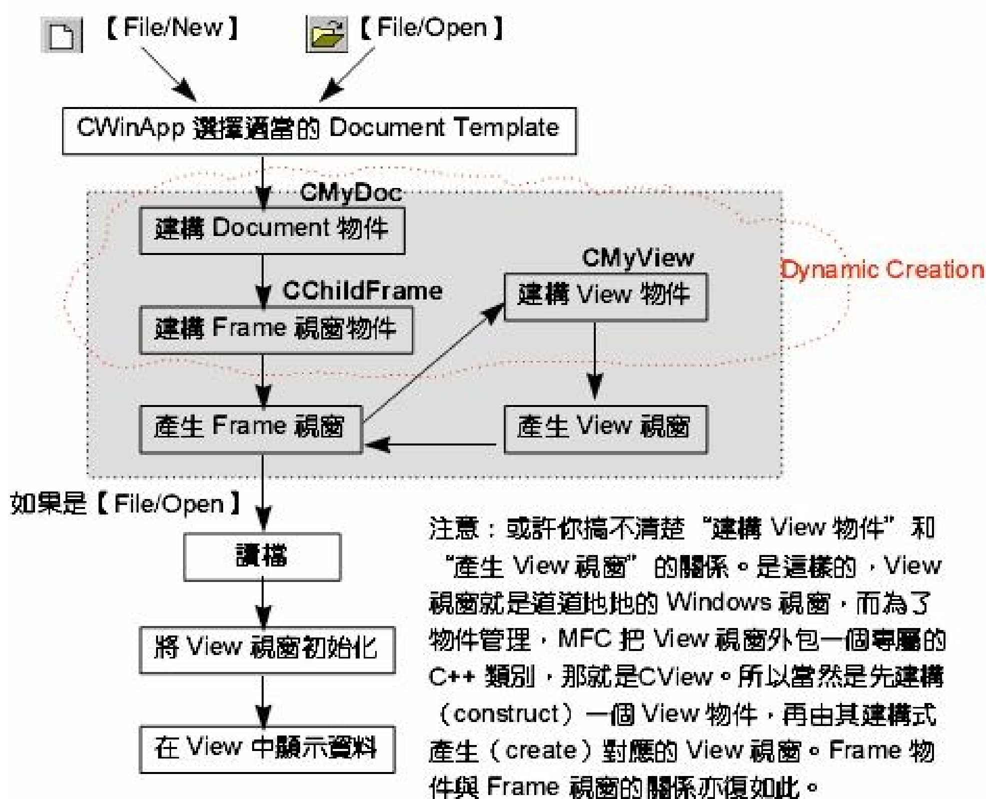


图8-1 Document/View/Frame 的产生

图8-1的灰色部份，正是Document Template动态产生 「三位一体之Document/View/Frame」的行动。下面的流程以及MFC原始代码足以澄清一切疑虑。在CMultiDocTemplate::OpenDocumentFile（注）出现之前的所有流程，我只做文字叙述，不显示其原始代码。本章稍后有一节「台面下的Serialize读文件奥秘」，则会将每一环节的原始代码呈现在你眼前，让你无所挂虑。

注：如果是 SDI 程序，那么就是 CSingleDocTemplate::OpenDocumentFile 被调用。但「多」比「单」有趣，而且本书范例Scribble 程序也使用 CMultiDocTemplate，所以我就以此为说明对象。

CSingleDocTemplate 只支持一种文件类型，所以它的成员变量是：

```
class CSingleDocTemplate : public CDocTemplate
{
...
protected:  // standard implementation
        CDocument* m_pOnlyDoc;
};
```

CMultiDocTemplate 支援多種文件型態，所以它的成員變數是：

```
class CMultiDocTemplate : public CDocTemplate
{
...
protected:  // standard implementation
        CPtrList m_docList;
};
```

当使用者选按【File/New】命令项，根据AppWizard为我们所做的Message Map，此一命令由CWinApp::OnFileNew 接手处理。后者调用 CDocManager::OnFileNew，后者再调用CWinApp::OpenDocumentFile，后者再调用CDocManager::OpenDocumentFile，后者再调用CMultiDocTemplate::OpenDocumentFile（这是观察MFC原始代码所得结果）：

```
// in AFXWIN.H
class CDocTemplate : public CCmdTarget
{
  ...
      UINT m_nIDResource;            // IDR_ for frame/menu/accel as well
      CRuntimeClass* m_pDocClass;     // class for creating new documents
      CRuntimeClass* m_pFrameClass;   // class for creating new frames
      CRuntimeClass* m_pViewClass;    // class for creating new views
      CString m_strDocStrings;        // '\n' separated names
  ...
}

// in DOCMULTI.CPP
CDocument* CMultiDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName,
      BOOL bMakeVisible)
{
      CDocument* pDocument = CreateNewDocument();
      ...
      CFrameWnd* pFrame = CreateNewFrame(pDocument, NULL);
      ...
      if (lpszPathName == NULL)
      {
            // create a new document - with default document name
            ...
      }
      else
      {
            // open an existing document
            ...
      }
      InitialUpdateFrame(pFrame, pDocument, bMakeVisible);
      return pDocument;
}
```

顾名思义，我们很容易作出这样的联想：CreateNewDocument动态产生 Document，CreateNewFrame 动态产生Document Frame。的确是这样没错，它们利用CRuntimeClass的CreateObject 做「动态生成」动作：

```
// in DOCTEMPL.CPP
```

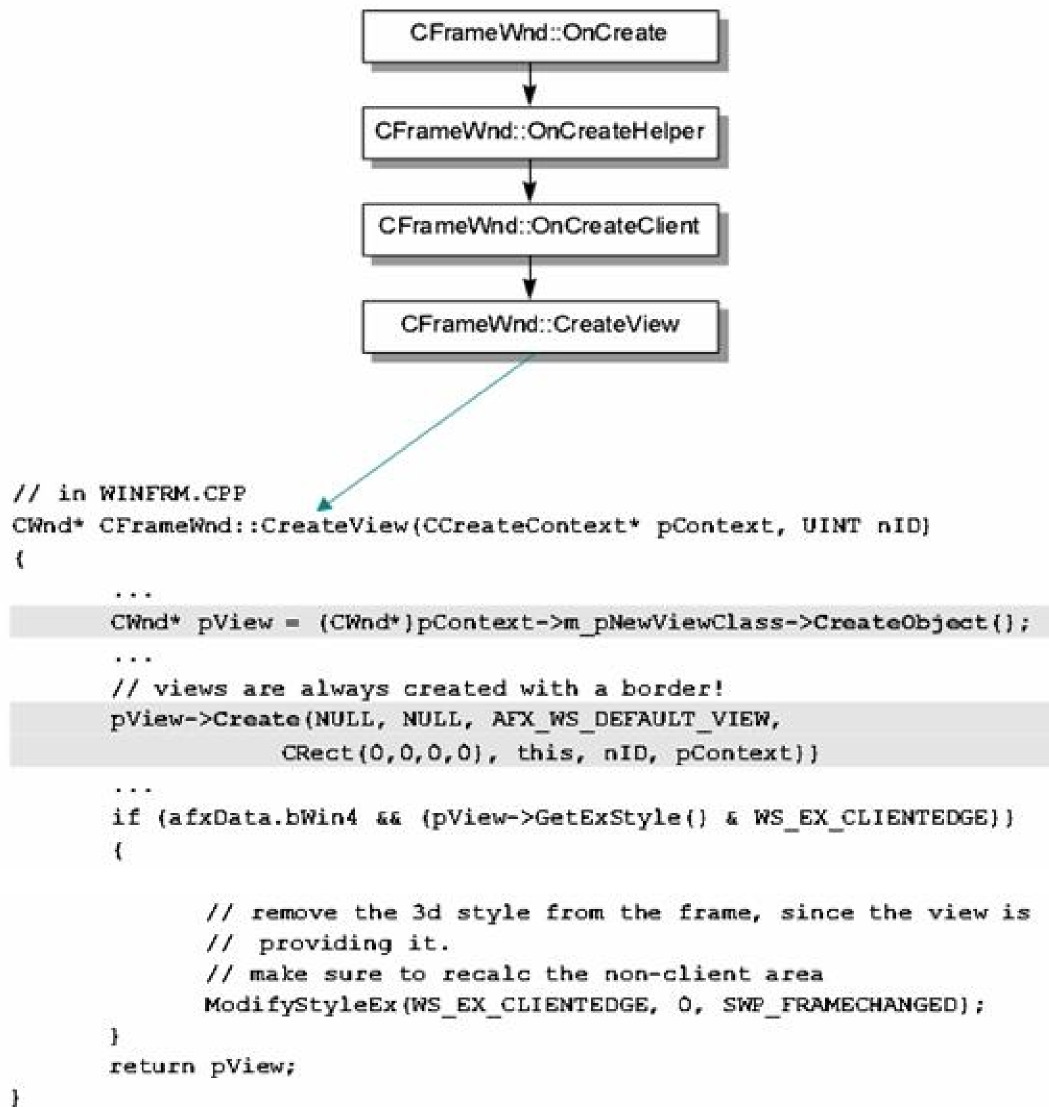
```
CDocument* CDocTemplate::CreateNewDocument()
{
        ...
        CDocument* pDocument = (CDocument*)m_pDocClass->CreateObject();
        ...
        AddDocument(pDocument);
        return pDocument;
}

CFrameWnd* CDocTemplate::CreateNewFrame(CDocument* pDoc, CFrameWnd* pOther)
{
        // create a frame wired to the specified document
        CCreateContext context;
        context.m_pCurrentFrame = pOther;
        context.m_pCurrentDoc = pDoc;
        context.m_pNewViewClass = m_pViewClass;
        context.m_pNewDocTemplate = this;
        ...
        CFrameWnd* pFrame = (CFrameWnd*)m_pFrameClass->CreateObject();
        ...
        // create new from resource
        pFrame->LoadFrame(m_nIDResource,
                WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE, // default frame styles
                NULL, &context)
        ...
        return pFrame;
}
```

在CreateNewFrame函数中，不仅Frame被动态生成出来了，其对应窗口也以LoadFrame产生出来了。但有两件事情令人不解。第一，我们没有看到View的动态生成动作；第二，出现一个奇怪的家伙CCreateContext，而前一个不解似乎能够着落到这个奇怪家伙的身上，因为CDocTemplate::m_pViewClass 被塞到它的一个字段中。

但是线索似乎已经中断，因为我们已经看不到任何可能的调用动作了。等一等！context 被用作LoadFrame的最后一个参数，这意味什么？还记得第六章「CFrameWnd::Create 产生主窗口（并先注册窗口类）」那一节提过 Create 的最后一个参数吗，正是这context。

那么，是不是Document Frame窗口产生之际由于WM_CREATE 的发生而刺激了什么动作？虽然其结果是正确的，但这样的联想也未免太天马行空了些。我只能说，经验累积出判断力！是的，WM_CREATE 引发CFrameWnd::OnCreate 被唤起，下面是相关的调用次序（经观察 MFC 原始代码而得知）：



不仅 View 对象被动态生成出来了，其对应的实际 Windows 窗口也以 Create 函数产生出来。

正因为MFC 把View 对象的动态生成动作包装得如此诡谲奇险，所以我才在图8-1中把「构造View对象」和「产生View窗口」这两个动作特别另立一旁：

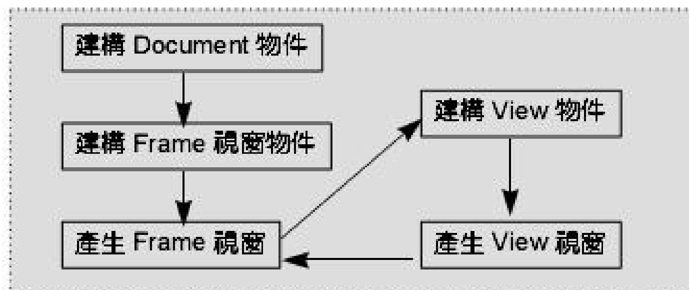


图8-2 解释CDocTemplate、CDocument、CView、CFrameWnd 之间的关系。下面则是一份文字整理：

- CWinApp 拥有一个对象指针：CDocManager* m_pDocManager。

- CDocManager拥有一个指针串列CPtrList m_templateList，用来维护一系列的Document Template。一个程序若支持两「种」文件类型，就应该有两份Document Templates，应用程序应该在CMyWinApp::InitInstance中以 AddDocTemplate将这些Document Templates加入由CDocManager所维护的串列之中。

- CDocTemplate拥有三个成员变量，分别持有Document、View、Frame 的CRumtimeClass 指针，另有一个成员变量m_nIDResource，用来表示此 Document显现时应该采用的UI对象。这四份数据应该在 CMyWinApp::InitInstance 函数构造CDocTemplate（注1）时指定之，成为构造函数的参数。当使用者欲打开一份文件（通常是借着【File/Open】或【File/New】命令项），CDocTemplate 即可藉由Document/View/Frame之CRuntimeClass 指针（注2）进行动态生成。

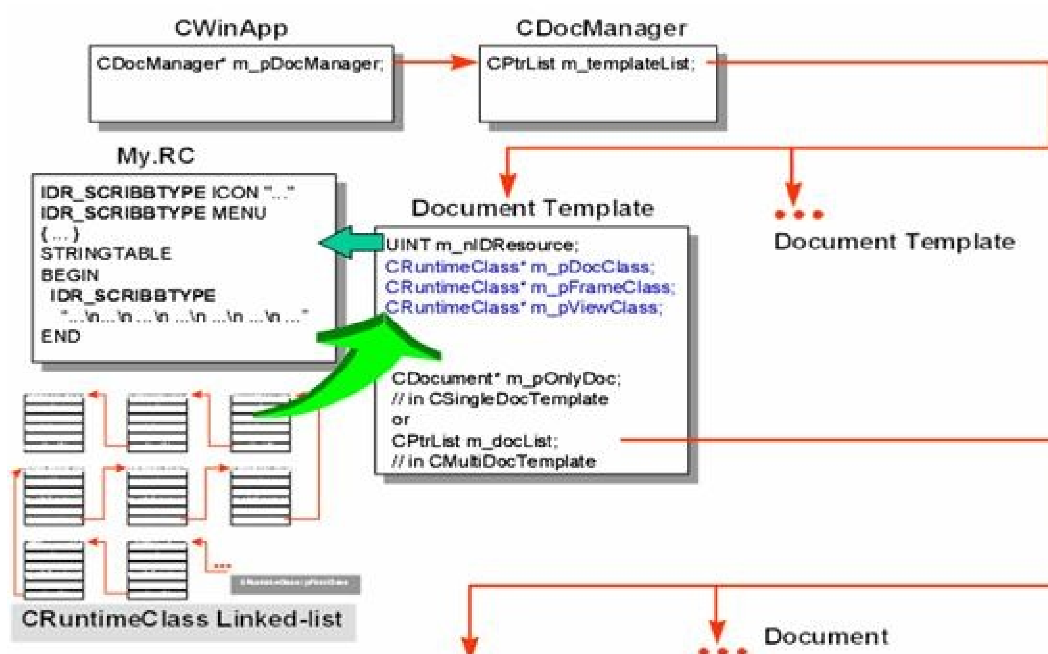
注1：在此我们必须有所选择，要不就使用CSingleDocTemplate，要不就使用CMultiDocTemplate，两者都是CDocTemplate的派生类。如果你选用CSingleDocTemplate，它有一个成员变量CDocument* m_pOnlyDoc，亦即它一次只能打开一份Document。如果你选用CMultiDocTemplate，它有一个成员变量 CPtrList m_docList，表示它能同时打开多个Documents。

注2：关于CRuntimeClass 与动态生成，我在第３章已经以DOS程序仿真之，本章稍后亦另有说明。

- CDocument有一个成员变量CDocTemplate* m_pDocTemplate，回指其 Document Template；另有一个成员变量CPtrList m_viewList，表示它可以同时维护一系列的Views。

- CFrameWnd有一个成员变量CView* m_pViewActive，指向目前正作用中的View。

- CView有一个成员变量CDocument* m_pDocument，指向相关的Document。



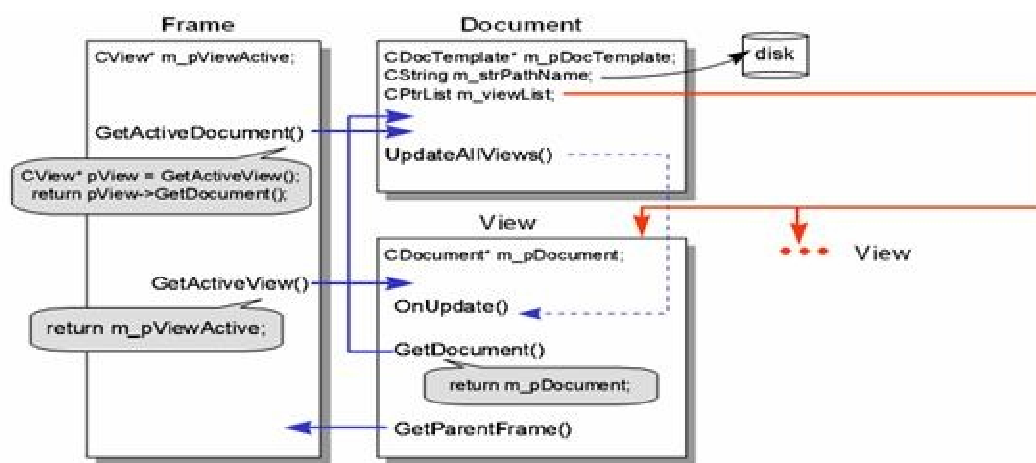


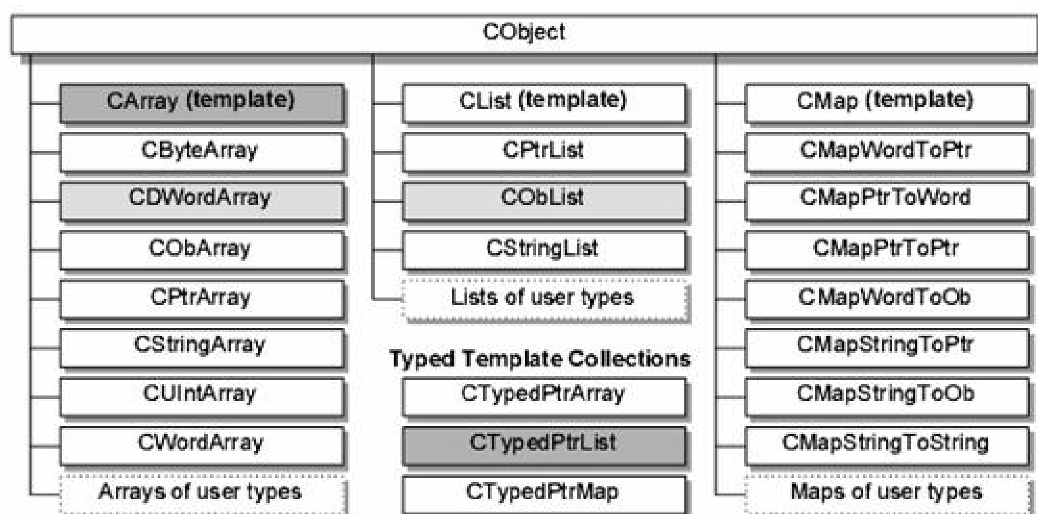
图8-2 CDocTemplate、CDocument、CView、CFrameWnd之间的关系

我把Document/View/Frame 的观念以狂风骤雨之势对你做了一个交待。模糊？晦暗？没有关系，马上我们就开始实现 Scribble Step1，你会从实现过程中慢慢体会上述观念。

Scribble Step1的Document——数据结构设计

Scribble 允许使用者在窗口中画图，画图的方式是以鼠标做为画笔，按下左键拖曳拉出线条。每次按下鼠标左键后一直到放开为止的连续坐标点构成线条（stroke）。整张图（整份文件）由线条构成，线条可由点、笔宽、笔色等等数据构成（但本例并无笔色数据）。

MFC的Collections Classes 中有许多适用于各种数据类型（如Byte、Word、DWord、Ptr）以及各种数据结构（如数组、串列）的现成类。如果我们尽可能把这些现成的类应用到程序的数据结构上面，就可以节省许多开发时间：



我们的设计最高原则就是尽量使用MFC已有的类，提高软件IC的重复使用性。上图浅色部分是Scribble 范例程序在16位MFC中采用的两个类。深色部分是Scribble范例程序在32位MFC中采用的两个类。

MFC Collection Classes 的选用

第 5 章末尾我曾经大致提过 MFC Collection Classes。它们分为三种类型，用来管理一大群对象：

- Array：数组，有次序性（需依序处理），可动态增减大小，索引值为整数。
- List：双向串列，有次序性（需依序处理），无索引。串列有头尾，可从头尾或从串列的任何位置安插元素，速度极快。
- Map：又称为Dictionary，其内对象成对存在，一为键值对象（key object），一为实值对象（value object）。

下面是其特性整理：

型態	有序	索引	插入元素	搜尋特定元素	複製元素
List	Yes	No	快	慢	可
Array	Yes	Yes（利用整數索引值）	慢	慢	可
Map	No	Yes（利用鍵值）	快	快	鍵值（key）不可複製，實值（value）可複製。

MFC Collection classes 所收集的对象中，有两种特别需要说明，一是Ob 一是Ptr：

- Ob 表示派生自CObject 的任何对象。MFC提供CObList、CObArray 两种类。
- Ptr表示对象指针。MFC 提供CPtrList、CPtrArray 两种类。

当我们考虑使用MFC collection classes 时，除了考虑上述三种类型的特性，还要考虑以下几点：

- 是否使用C++ template（对于type-safe 极有帮助）。
- 储存于collection class 之中的元素是否要做文件读写动作（Serialize）。
- 储存于collection class 之中的元素是否要有倾印（dump）和错误诊断能力。

下表是对所有collection classes 性质的一份摘要整理（参考自微软的官方手册：Programming With MFC and Win32）：

類別	C++ template	Serializable	Dumpable	type-safe
CArray	Yes	Yes ①	Yes ①	No
CTypedPtrArray	Yes	Depends ②	Yes	Yes
CByteArray	No	Yes	Yes	Yes ③
CDWordArray	No	Yes	Yes	Yes ③
CObArray	No	Yes	Yes	No
CPtrArray	No	No	Yes	No
CStringArray	No	Yes	Yes	Yes ③
CWordArray	No	Yes	Yes	Yes ③
CUIntArray	No	No ④	Yes	Yes ③
CList	Yes	Yes ①	Yes ①	No
CTypedPtrList	Yes	Depends ②	Yes	Yes
CObList	No	Yes	Yes	No
CPtrList	No	No	Yes	No
CStringList	No	Yes	Yes	Yes ③
CMap	Yes	Yes ①	Yes ①	No
CTypedPtrMap	Yes	Depends ②	Yes	Yes
CMapPtrToWord	No	No	Yes	No
CMapPtrToPtr	No	No	Yes	No
CMapStringToOb	No	Yes	Yes	No
CMapStringToPtr	No	No	Yes	No
CMapStringToString	No	Yes	Yes	Yes ③
CMapWordToOb	No	Yes	Yes	No
CMapWordToPtr	No	No	Yes	No

① 若要文件读写，你必须明白调用collection object的Serialize 函数；若要内容倾印，你必须明白调用其 Dump 函数。不能够使用 `archive << obj` 或 `dmp << obj` 这种型式。

②究竟是否 Serializable，必须视其内含对象而定。举个例，如果一个typedpointerarray 是以 CObArray 为基础，那么它是 Serializable；如果它是以 CPtrArray 为基础，那么它就不是 Serializable。一般而言，Ptr 都不能够被 Serialized。

③ 虽然它是 non-template，但如果照预定计划去使用它（例如以 CByteArray 储存 bytes，而不是用来储存 char），那么它还是 type-safe 的。

④ 手册上说它并非 Serializable，但我存疑。各位不妨试验之。

Template-Based Classes

本书第 2 章末尾已经介绍过所谓的 C++ template。MFC 的 collection classes 里头有一些是 template-based，对于类型检验的功夫做得比较好。这些类区分为：

简单型——CArray、CList、CMap。它们都派生自CObject，所以它们都具备了文件读写、执行时期型别鉴识、动态生成等性质。

类型指针型——CTypedPtrArray、CTypedPtrList、CTypedPtrMap。这些类要求你在参数中指定基类，而基类必须是MFC之中的 non-template pointer collections，例如CObList或CPtrArray。你的新类将继承基类的所有性质。

Template-Based Classes 的使用方法（注意：需含入 afxtempl.h，如 p.903 stdafx.h）

简单型 template-based classes 使用时需要指定参数：

- `CArray<TYPE, ARG_TYPE>`
- `CList<TYPE, ARG_TYPE>`
- `CMap<KEY, ARG_KEY, VALUE, ARG_VALUE>`

其中TYPE用来指定你希望收集的对象的类型，它们可以是：

- C++ 基础型别，如 int、char、long、float 等等。
- C++ 结构或类。

ARG_TYPE 则用来指定函数的参数类型。举个例，下面程序代码表示我们需要一个int 阵列，数组成员函数（例如 Add）的参数是int：

```
CArray<int, int> m_intArray;
m_intArray.Add(15);
```

再举一例，下面程序代码表示我们需要一个由int组成的串列，串列成员函数（例如AddTail）的参数是int：

```
CList<int, int> m_intList;
m_intList.AddTail(36);
m_intList.RemoveAll();
```

再举一例，下面程序代码表示我们需要一个由CPoint组成的数组，数组成员函数（例如Add）的参数是CPoint：

```
CArray<CPoint, CPoint> m_pointArray;
CPoint point(18, 64);
m_pointArray.Add(point);
```

「类型指针」型的template-based classes使用时亦需指定参数：

```
CTypedPtrArray<BASE_CLASS, TYPE>
CTypedPtrList<BASE_CLASS, TYPE>
CTypedPtrMap<BASE_CLASS, KEY, VALUE>
```

其中TYPE用来指定你希望收集的对象的类型，它们可以是：

- C++ 基础型别，如 int、char、long、float 等等。
- C++ 结构或类。

BASE_CLASS 则用来指定基类，它可以是任何用来收集指针的 non-template collection classes，例如CObList或CObArray或CPtrList或CPtrArray等等。举个例子，下面程序代码表示我们需要一个派生自CObList 的类，用来管理一个串列，而串列组成份子为CStroke*：

```
CTypedPtrList<CObList,CStroke*> m_strokeList;
CStroke* pStrokeItem = new CStroke(20);
m_strokeList.AddTail(pStrokeItem);
```

CScribbleDoc 的修改

了解了Collection Classes中各类的特性以及所谓 template/nontemplate 版本之后，以本例之情况而言，很显然：

不定量的线条数可以利用串列（linked list）来表示，那么MFC的CObList 恰可用来表现这样的串列。CObList 规定其每个元素必须是一个「CObject 派生类」的对象实体，好啊，没问题，我们就设计一个名为CStroke 的类，派生自CObject，代表一条线条。为了type-safe，我们选择template 版本，所以设计出这样的Document：

```
class CScribbleDoc : public CDocument
{
...
public:
        CTypedPtrList<CObList,CStroke*>  m_strokeList;
...
}
```

线条由笔宽和坐标点构成，所以CStroke应该有m_nPenWidth 成员变量，但一长串的坐标点以什么来管理好呢？数组是个不错的选择，至于数组内要放什么类型的数据，我们不妨先着一鞭，想想这些坐标是怎么获得的。这些坐标显然是在鼠标左键按下时进入程序之中，也就是利用OnLButtonDown函数的参数 CPoint。CPoint符合前一节所说的数组元素类型条件，所以CStroke的成员变量可以这么设计：

```
class CStroke : public CObject
{
...
protected:
        UINT m_nPenWidth;
public:
        CArray<CPoint,CPoint>  m_pointArray;
...
}
```

至于CPoint实际内容是什么，就甭管了吧。

事实上CPoint是一个由两个long组成的结构，两个long各代表x和y坐标。

CScribble Step1 Document：（本图为了说明方便，以 CObList 代替实际使用之 CTypedPtrList）

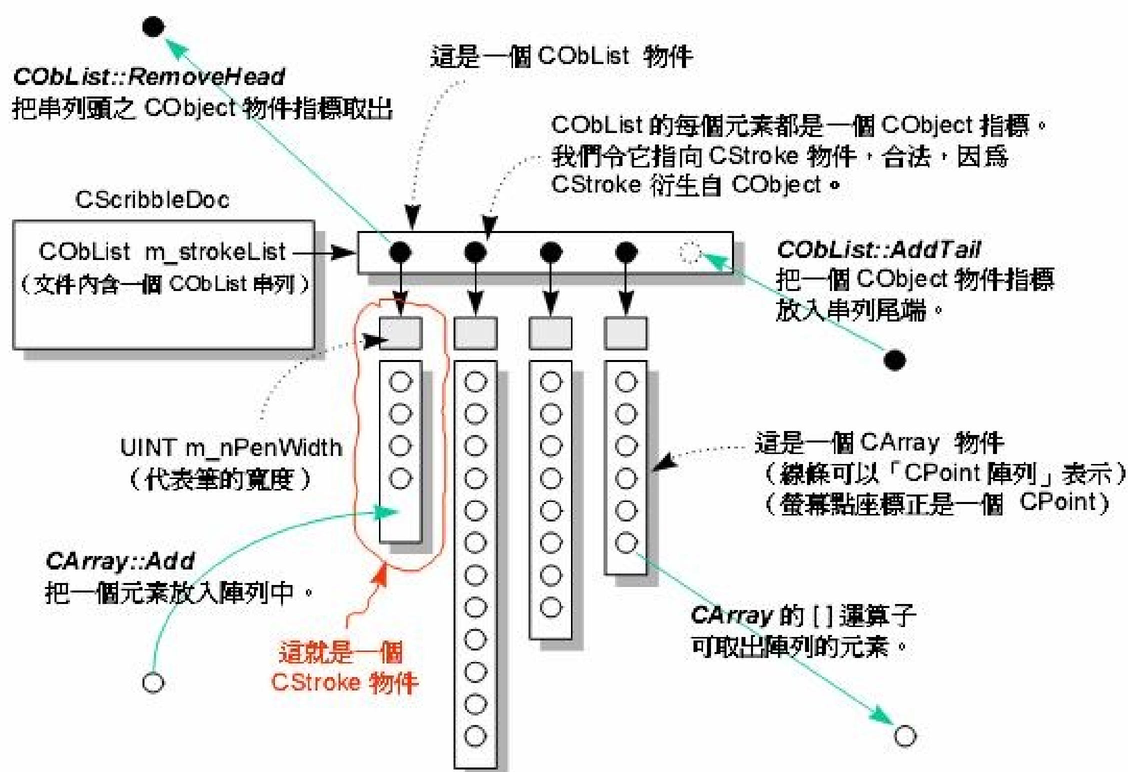


图8-3a Scribble Step1的文件由线条构成，线条又由点数组构成

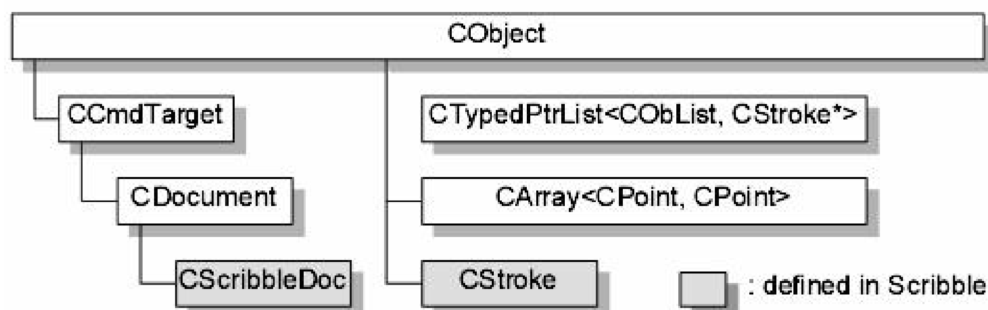


图8-3b Scribble Step1文件所使用的类

CScribbleDoc 内嵌一个CObList对象，CObList串列中的每个元素都是一个CStroke对象指针，而CStroke又内嵌一个CArray 对象。下面是Step1程序的 Document 设计。

SCRIBBLEDOC.H（阴影表示与 Step0 的差异）

```
#0001 ///////////////////////////////////////////////////////////////////////
#0002 // class CStroke
#0003 //
#0004 // A stroke is a series of connected points in the scribble drawing.
#0005 // A scribble document may have multiple strokes.
#0006
#0007 class CStroke : public CObject
#0008 {
#0009 public:
#0010         CStroke(UINT nPenWidth);
#0011
#0012 protected:
#0013         CStroke();
#0014         DECLARE_SERIAL(CStroke)
#0015
#0016 // Attributes
#0017 protected:
#0018         UINT   m_nPenWidth;    // one pen width applies to entire stroke
#0019 public:
#0020         CArray<CPoint,CPoint>  m_pointArray;   // series of connected
points
#0021
#0022 // Operations
#0023 public:
#0024         BOOL DrawStroke(CDC* pDC);
#0025
#0026 public:
#0027         virtual void Serialize(CArchive& ar);
#0028 };
#0029
#0030 ///////////////////////////////////////////////////////////////////////
#0031
#0032 class CScribbleDoc : public CDocument
#0033 {
#0034 protected: // create from serialization only
#0035         CScribbleDoc();
#0036         DECLARE_DYNCREATE(CScribbleDoc)
#0037
#0038 // Attributes
#0039 protected:
```

```
#0040         // The document keeps track of the current pen width on
#0041         // behalf of all views. We'd like the user interface of
#0042         // Scribble to be such that if the user chooses the Draw
#0043         // Thick Line command, it will apply to all views, not just
#0044         // the view that currently has the focus.
#0045
#0046         UINT   m_nPenWidth;    // current user-selected pen width
#0047         CPen   m_penCur;      // pen created according to
#0048                              // user-selected pen style (width)
#0049  public:
#0050         CTypedPtrList<CObList,CStroke*>    m_strokeList;
#0051         CPen*         GetCurrentPen() { return &m_penCur; }
#0052
#0053  // Operations
#0054  public:
#0055         CStroke* NewStroke();
```

```
#0056
#0057  // Overrides
#0058         // ClassWizard generated virtual function overrides
#0059         //{{AFX_VIRTUAL(CScribbleDoc)
#0060         public:
#0061         virtual BOOL OnNewDocument();
#0062         virtual void Serialize(CArchive& ar);
#0063         virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
#0064         virtual void DeleteContents();
#0065         //}}AFX_VIRTUAL
#0066
#0067  // Implementation
#0068  public:
#0069         virtual ~CScribbleDoc();
#0070  #ifdef _DEBUG
#0071         virtual void AssertValid() const;
#0072         virtual void Dump(CDumpContext& dc) const;
#0073  #endif
#0074
#0075  protected:
#0076         void  InitDocument();
#0077
#0078  // Generated message map functions
#0079  protected:
#0080    //{{AFX_MSG(CScribbleDoc)
#0081      // NOTE - the ClassWizard will add and remove member functions here.
#0082      //    DO NOT EDIT what you see in these blocks of generated code !
#0083    //}}AFX_MSG
#0084    DECLARE_MESSAGE_MAP()
#0085  };
```

如果你把本书第一版（使用VC++ 4.0）的Scribble step1原封不动地在 VC++ 4.2或VC++ 5.0中编译，你会获得好几个编译错误。问题出在 SCRIBBLEDOC.H 文件：

```
// forward declaration of data structure class
class CStroke;

class CScribbleDoc : public CDocument
{
   ...
};

class CStroke : public CObject
{
   ...
};
```

并不是程序设计上有什么错误，你只要把CStroke的声明由CScribbleDoc 之后搬移到CScribbleDoc 之前即可。由此观之，VC++ 4.2和VC++ 5.0 的编译器似乎不支持forward declaration。真是没道理！

SCRIBBLEDOC.CPP（阴影表示与 Step0 的差异）

```
#0001  #include "stdafx.h"
#0002  #include "Scribble.h"
#0003
#0004  #include "ScribbleDoc.h"
#0005
#0006  #ifdef _DEBUG
#0007  #define new DEBUG_NEW
#0008  #undef THIS_FILE
#0009  static char THIS_FILE[] = __FILE__;
#0010  #endif
#0011
#0012  /////////////////////////////////////////////////////////////////////////////
#0013  // CScribbleDoc
#0014
#0015  IMPLEMENT_DYNCREATE(CScribbleDoc, CDocument)
#0016
#0017  BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
#0018         //{{AFX_MSG_MAP(CScribbleDoc)
#0019             // NOTE - the ClassWizard will add and remove mapping macros here.
#0020             // DO NOT EDIT what you see in these blocks of generated code!
#0021         //}}AFX_MSG_MAP
#0022  END_MESSAGE_MAP()
#0023
#0024  /////////////////////////////////////////////////////////////////////////////
```

```
#0025 // CScribbleDoc construction/destruction
#0026
#0027 CScribbleDoc::CScribbleDoc()
#0028 {
#0029         // TODO: add one-time construction code here
#0030
#0031 }
#0032
#0033 CScribbleDoc::~CScribbleDoc()
#0034 {
#0035 }
#0036
#0037 BOOL CScribbleDoc::OnNewDocument()
#0038 {
#0039         if (!CDocument::OnNewDocument())
#0040                 return FALSE;
#0041         InitDocument();
#0042         return TRUE;
#0043 }
#0044
#0045 /////////////////////////////////////////////////////////////////////////////
#0046 // CScribbleDoc serialization
#0047
#0048 void CScribbleDoc::Serialize(CArchive& ar)
#0049 {
#0050         if (ar.IsStoring())
#0051         {
#0052         }
#0053         else
#0054         {
#0055         }
#0056         m_strokeList.Serialize(ar);
#0057 }
#0058
#0059 /////////////////////////////////////////////////////////////////////////////
#0060 // CScribbleDoc diagnostics
#0061
#0062 #ifdef _DEBUG
#0063 void CScribbleDoc::AssertValid() const
#0064 {
#0065         CDocument::AssertValid();
#0066 }
#0067
#0068 void CScribbleDoc::Dump(CDumpContext& dc) const
#0069 {
#0070         CDocument::Dump(dc);
#0071 }
#0072 #endif //_DEBUG
```

```
#0073
#0074 ///////////////////////////////////////////////////////////////////
#0075 // CScribbleDoc commands
#0076
#0077 BOOL CScribbleDoc::OnOpenDocument(LPCTSTR lpszPathName)
#0078 {
#0079         if (!CDocument::OnOpenDocument(lpszPathName))
#0080                 return FALSE;
#0081         InitDocument();
#0082         return TRUE;
#0083 }
#0084
#0085 void CScribbleDoc::DeleteContents()
#0086 {
#0087         while (!m_strokeList.IsEmpty())
#0088         {
#0089                 delete m_strokeList.RemoveHead();
#0090         }
#0091         CDocument::DeleteContents();
#0092 }
#0093
#0094 void CScribbleDoc::InitDocument()
#0095 {
#0096         m_nPenWidth = 2; // default 2 pixel pen width
#0097         // solid, black pen
#0098         m_penCur.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0));
#0099 }
#0100
#0101 CStroke* CScribbleDoc::NewStroke()
#0102 {
#0103         CStroke* pStrokeItem = new CStroke(m_nPenWidth);
#0104         m_strokeList.AddTail(pStrokeItem);
#0105         SetModifiedFlag();  // Mark the document as having been modified, for
#0106                             // purposes of confirming File Close.
#0107         return pStrokeItem;
#0108 }
#0109
#0110
#0111
#0112
#0113 ///////////////////////////////////////////////////////////////////
#0114 // CStroke
#0115
#0116 IMPLEMENT_SERIAL(CStroke, CObject, 1)
#0117 CStroke::CStroke()
#0118 {
#0119         // This empty constructor should be used by serialization only
#0120 }
#0121
#0122 CStroke::CStroke(UINT nPenWidth)
#0123 {
#0124         m_nPenWidth = nPenWidth;
#0125 }
#0126
#0127 void CStroke::Serialize(CArchive& ar)
#0128 {
#0129         if (ar.IsStoring())
#0130         {
#0131                 ar << (WORD)m_nPenWidth;
#0132                 m_pointArray.Serialize(ar);
#0133         }
#0134         else
#0135         {
#0136                 WORD w;
#0137                 ar >> w;
#0138                 m_nPenWidth = w;
```

```
#0139                m_pointArray.Serialize(ar);
#0140        }
#0141 }
#0142
#0143 BOOL CStroke::DrawStroke(CDC* pDC)
#0144 {
#0145        CPen penStroke;
#0146        if (!penStroke.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0)))
#0147                return FALSE;
#0148        CPen* pOldPen = pDC->SelectObject(&penStroke);
#0149        pDC->MoveTo(m_pointArray[0]);
#0150        for (int i=1; i < m_pointArray.GetSize(); i++)
#0151        {
#0152                pDC->LineTo(m_pointArray[i]);
#0153        }
#0154
#0155        pDC->SelectObject(pOldPen);
#0156        return TRUE;
#0157 }
```

为了了解线条的产生经历了哪些成员函数，使用了哪些成员变量，我把图 8-3所显示的各类的成员整理于下。让我们以top-down的方式看看文件组成份子的运作。

文件：一连串的线条

Scribble 文件本身由许多线条组合而成。而你知道，以串列（linked list）表示不定个数的东西最是理想了。MFC有没有现成的「串列」类呢？有，CObList 就是。它的每一个元素都必须是CObject*。回想一下我在第二章介绍的「职员」例子：

我们有一个职员串列，串列的每一个元素的类型是「指向最基类之指针」。如果基类有一个「计薪」方法（虚函数），那么我们就可以一个「一般性」的循环把串列巡访一遍；巡到不同的职员型别，就调用该型别的计薪方法。

如今我们选用CObList，情况不就和上述职员例子如出一辙吗？CObject 的许多好性质，如 Serialization、RTTI、Dynamic Creation，可以非常简便地应用到我们极为「一般性」的操作上。这一点在稍后的 Serialization 动作上更表现得淋漓尽致。

CScribbleDoc 的成员变量

- m_strokeList：这是一个CObList 对象，代表一个串列。串列中的元素是什么类型？答案是CObject*。但实际运作时，我们可以把基类之指针指向派生类之对象（还记得第 2 章我介绍虚函数时特别强调的吧）。现在我们想让这个串列成为「由CStroke 对象构成的串列」，因此显然CStroke 必须派生自CObject才行，而事实上它的确是。
- m_nPenWidth：每一线条都有自己的笔宽，而目前使用的笔宽记录于此。
- m_penCur：这是一个CPen 对象。程序依据上述的笔宽，配置一支笔，准备用来画线条。笔宽可以指定，但那是第10 章的事。注意，笔宽的设定对象是线条，不是单一的点，也不是一整张图。

CObList

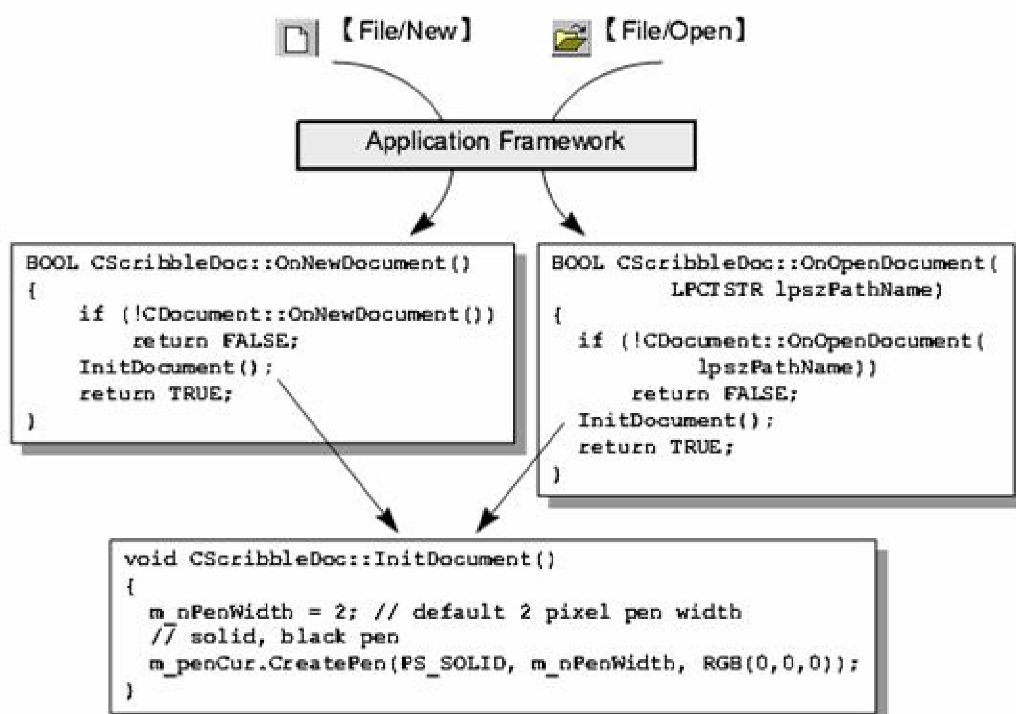
这是MFC的内建类，提供我们串列服务。串列的每个元素都必须是 CObject*。本处将用到四个成员函数：

- AddTail：在串列尾端加上一个元素。
- IsEmpty：串列是否为空？
- RemoveHead：把串列整个拿掉。
- Serialize：文件读写。这是个空的虚函数，改写它正是我们稍后要做的努力。

CScribbleDoc 的成员函数

- OnNewDocument、OnOpenDocument、InitDocument。产生Document 的时机有二，一是使用者选按【File/New】，一是使用者选按【File/Open】。当这两种情况发生，Application Framework 会分别调用 Document 类的 OnNewDocument 和 OnOpenDocument。为了应用程序本身的特性考虑（例如本例画笔的产生以及笔宽的设定），我们应该改写这些虚函数。

  本例把文件初始化工作（画笔以及笔宽的设定）分割出来，独立于 InitDocument 函数中，因此上述的OnNew 和*OnOpen* 两函数都调用 InitDocument。



- NewStroke。这个函数将产生一个新的 CStroke对象，并把它加到串列之中。 很显然这应该在鼠标左键按下时发生（我们将在 CScribbleView 之中处理鼠标消息）。本函数动作如下：

```
CStroke* CScribbleDoc::NewStroke()
{
    CStroke* pStrokeItem = new CStroke(m_nPenWidth);
    m_strokeList.AddTail(pStrokeItem);
    SetModifiedFlag();  // Mark the document as having been modified, for
                        // purposes of confirming File Close.
    return pStrokeItem;
}
```

这就产生了一个新线条，设定了线条宽度，并将新线条加入串列尾端。

- DeleteContent。利用 CObList::RemoveHead 把串列的最前端元素拿掉。

```
void CScribbleDoc::DeleteContents()
{
        while (!m_strokeList.IsEmpty())
        {
                delete m_strokeList.RemoveHead();
        }
        CDocument::DeleteContents();
}
```

- Serialize。这个函数负责文件读写。由于文件掌管线条串列，线条串列又掌管各线条，我们可以善用这些阶层关系：

```
void CScribbleDoc::Serialize(CArchive& ar)
{
        if (ar.IsStoring())
        {
        }
        else
        {
        }
        m_strokeList.Serialize(ar);
}
```

我们有充份的理由认为，CObList::Serialize 的内部动作，一定是以一个循环巡访所有的元素，一一调用各元素（是个指针）所指向的对象的 Serialize 函数。就好像第 2 章「职员」串列中的计薪方法一样。

马上我们就会看到，Serialize 如何层层下达。那是很深入的探讨，你要先有心理准备。

线条与坐标点

Scribble 的文件数据由线条构成，线条又由点数组构成，点又由（x,y）坐标构成。我们将设计CStroke用以描述线条，并直接采用MFC的CArray描述点数组。

CStroke 的成员变量

- m_pointArray：这是一个 CArray 对象，用以记录一系列的CPoint对象，这些CPoint 对象由鼠标坐标转化而来。
- m_nPenWidth：一个整数，代表线条宽度。虽然Scribble Step1的线条宽度是固定的，但第10章允许改变宽度。

`CArray<CPoint, CPoint>`

CArray 是MFC内建类，提供数组的各种服务。本例利用其template性质，指定数组内容为CPoint。本例将用到CArray 的两个成员函数和一个运算符：

- GetSize：取得数组中的元素个数。
- Add：在数组尾端增加一个元素。必要时扩大数组的大小。这个动作会在鼠标左键按下后被持续调用，请看ScribbleView::OnLButtonDown。
- operator[ ]：以指定之索引值取得或设定数组元素内容。

它们的详细规格请参考 MFC Class Library Reference。

CStroke 的成员函数

- DrawStroke ：绘图原本是View 的责任，为什么却在CStroke中有一个DrawStroke？因为线条的内容只有CStroke自己知道，当然由CStroke 的成员函数把它画出来最是理想。这么一来，View 就可以一一调用线条自己的绘图函数，很轻松。

  此函数把点坐标从数组之中一个一个取出，画到窗口上，所以你会看到整个原始绘图过程的重现，而不是一整张图啪一下子出现。想当然耳，这个函数内会有CreatePen、SelectObject、MoveTo、LineTo等GDI动作，以及从数组中取坐标点的动作。取点动作直接利用CArray的operator[ ]运算符即可办到：

```
BOOL CStroke::DrawStroke(CDC* pDC)
{
        CPen penStroke;
        if (!penStroke.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0)))
                return FALSE;
        CPen* pOldPen = pDC->SelectObject(&penStroke);
        pDC->MoveTo(m_pointArray[0]);
        for (int i=1; i < m_pointArray.GetSize(); i++)
        {
                pDC->LineTo(m_pointArray[i]);
        }
        pDC->SelectObject(pOldPen);
        return TRUE;
}
```

- Serialize：让我们这么想象写文件动作：使用者下命令给程序，程序发命令给文件，文件发命令给线条，线条发命令给点数组，点数组于是把一个个的坐标点写入磁盘中。请注意，每一线条除了拥有点数组之外，还有一个笔划宽度，读写文件时可不要忘了这份数据。

```
void CStroke::Serialize(CArchive& ar)
{
        if (ar.IsStoring())
        {  // 寫檔
                ar << (WORD)m_nPenWidth;
                m_pointArray.Serialize(ar);
        }
        else
        {  // 讀檔
                WORD w;
                ar >> w;
                m_nPenWidth = w;
                m_pointArray.Serialize(ar);
        }
}
```

肯定你会产生两个疑问：

1.为什么点数组的读文件写文件动作完全一样，都是Serialize(ar)呢？

2. 线条串列的Serialize 函数如何能够把命令交派到线条的Serialize 函数呢？

第一个问题的答案很简单，第二个问题的答案很复杂。稍后我对此有所解释。

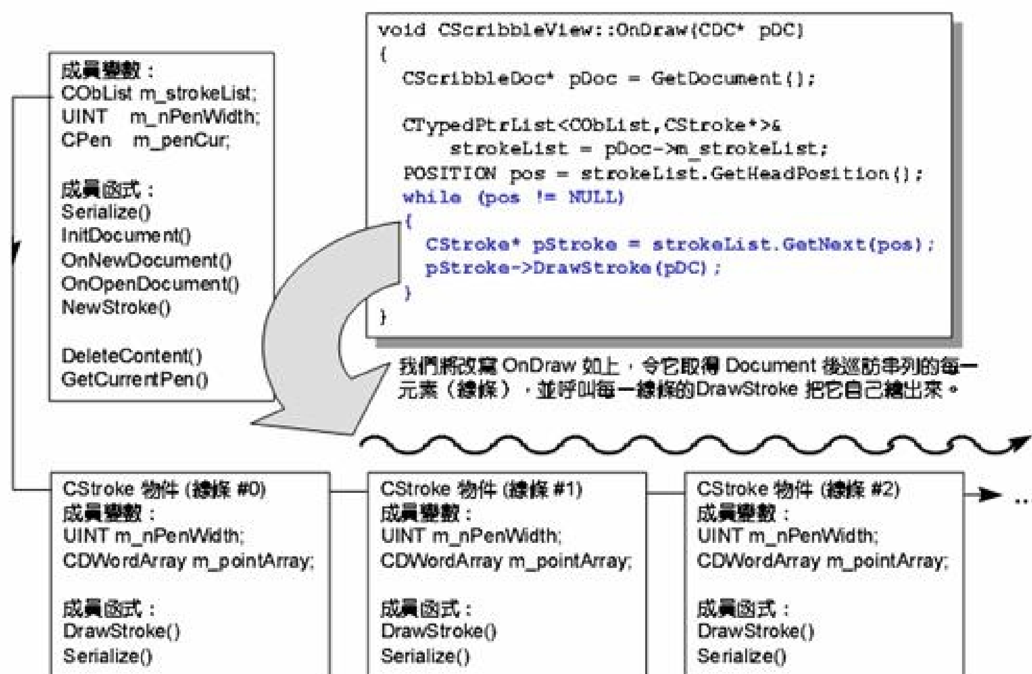


图8-4 Scribble的Document/View成员鸟瞰

图8-4 把Scribble Step1的Document/View重要成员集中在一起显示，帮助你做大局观。请注意，虽然本图把「成员函数」和「成员变量」画在每一个对象之中，但你知道，事实上C++类的成员函数另放在对象内存以外，并不是每一个对象都有一份函数代码。只有non-static成员变量，才会每个对象各有一份。这个观念我曾在第 2 章强调过。

Scribble Step1 的 View：数据重绘与编辑

View 有两个最重要的任务，一是负责数据的显示，另一是负责数据的编辑（透过键盘或鼠标）。本例的 CScribbleView 包括以下特质：

- 解读CScribbleDoc中的数据，包括笔宽以及一系列的CPoint对象，画在 View窗口上。
- 允许使用者以鼠标左键充当画笔在View窗口内涂抹，换句话说 CSribbleView必须接受并处理WM_LBUTTONDOWN、WM_MOUSEMOVE、WM_LBUTTONUP三个消息。

当Framework收到WM_PAINT，表示画面需要重绘，它会调用OnDraw（注），由OnDraw 执行真正的绘图动作。什么时候会产生重绘消息WM_PAINT 呢？当使用者改变窗口大小，或是将窗口图标化之后再恢复原状，或是来自程序（自己或别人）刻意的制造。除了在必须重绘时重绘之外，做为一个绘图软件，Scribble 还必须「实时」反应鼠标左键在窗口上移动的轨迹，不能等到WM_PAINT 产生了才有所反应。所以，我们必须在OnMouseMove 中也做绘图动作，那是针对一个点一个点的绘图，而 OnDraw 是大规模的全部重绘。

注：其实Framework是先调用OnPaint，OnPaint再调用OnDraw。关于 OnPaint，第12 章谈到打印机时再说。

绘图前当然必须获得数据内容，调用GetDocument即可获得，它传回一个 CScribbleDoc对象指针。别忘了View 和Document 以及Frame窗口早在注册 Document Template 时就建立彼此间的关联了。所以，从CScribbleView 发出的GetDocument 函数当然能够获得CScribbleDoc的对象指针。View可以藉此指针取得Document的数据，然后显示。

CScribbleView 的修改

以下是Step1程序的View 的设计。其中有鼠标接口，也有数据显示功能OnDraw。

SCRIBBLEVIEW.H（阴影表示与 Step0 的差异）

```
#0001 class CScribbleView : public CView
#0002 {
#0003 protected: // create from serialization only
#0004         CScribbleView();
#0005         DECLARE_DYNCREATE(CScribbleView)
#0006
#0007 // Attributes
#0008 public:
#0009         CScribbleDoc* GetDocument();
#0010
#0011 protected:
#0012         CStroke*  m_pStrokeCur;  // the stroke in progress
#0013         CPoint    m_ptPrev; // the last mouse pt in the stroke in progress
#0014
#0015 // Operations
#0016 public:
#0017
#0018 // Overrides
#0019         // ClassWizard generated virtual function overrides
#0020         //{{AFX_VIRTUAL(CScribbleView)
#0021         public:
#0022         virtual void OnDraw(CDC* pDC);  // overridden to draw this view
#0023         virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
#0024         protected:
#0025         virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
#0026         virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
#0027         virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
#0028         //}}AFX_VIRTUAL
#0029
#0030 // Implementation
#0031 public:
#0032         virtual ~CScribbleView();

#0033 #ifdef _DEBUG
#0034         virtual void AssertValid() const;
#0035         virtual void Dump(CDumpContext& dc) const;
#0036 #endif
#0037
#0038 protected:
#0039

#0040 // Generated message map functions
#0041 protected:
#0042         //{{AFX_MSG(CScribbleView)
#0043         afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
#0044         afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
#0045         afx_msg void OnMouseMove(UINT nFlags, CPoint point);
#0046         //}}AFX_MSG
#0047         DECLARE_MESSAGE_MAP()
#0048 };
#0049
#0050 #ifndef _DEBUG  // debug version in ScribVw.cpp
#0051 inline CScribbleDoc* CScribbleView::GetDocument()
#0052    { return (CScribbleDoc*)m_pDocument; }
#0053 #endif
```

SCRIBBLEVIEW.CPP（阴影表示与Step0的差异）

```
#0001  #include "stdafx.h"
#0002  #include "Scribble.h"
#0003
#0004  #include "ScribbleDoc.h"
#0005  #include "ScribbleView.h"
#0006
#0007  #ifdef _DEBUG
#0008  #define new DEBUG_NEW
#0009  #undef THIS_FILE
#0010  static char THIS_FILE[] = __FILE__;
#0011  #endif
#0012
#0013  /////////////////////////////////////////////////////////////////////
#0014  // CScribbleView
#0015
#0016  IMPLEMENT_DYNCREATE(CScribbleView, CView)
#0017
#0018  BEGIN_MESSAGE_MAP(CScribbleView, CView)
#0019          //{{AFX_MSG_MAP(CScribbleView)
#0020          ON_WM_LBUTTONDOWN()
#0021          ON_WM_LBUTTONUP()
#0022          ON_WM_MOUSEMOVE()
#0023          //}}AFX_MSG_MAP
#0024          // Standard printing commands
#0025          ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
#0026          ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
#0027          ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
#0028  END_MESSAGE_MAP()

#0029
#0030  /////////////////////////////////////////////////////////////////////
#0031  // CScribbleView construction/destruction
#0032
#0033  CScribbleView::CScribbleView()
#0034  {
#0035          // TODO: add construction code here
#0036
#0037  }
#0038
#0039  CScribbleView::~CScribbleView()
#0040  {
#0041  }
#0042
#0043  BOOL CScribbleView::PreCreateWindow(CREATESTRUCT& cs)
#0044  {
#0045          // TODO: Modify the Window class or styles here by modifying
#0046          //  the CREATESTRUCT cs
#0047
#0048          return CView::PreCreateWindow(cs);
#0049  }
#0050
#0051  /////////////////////////////////////////////////////////////////////
#0052  // CScribbleView drawing
#0053
#0054  void CScribbleView::OnDraw(CDC* pDC)
#0055  {
#0056          CScribbleDoc* pDoc = GetDocument();
#0057          ASSERT_VALID(pDoc);
#0058
```

```
#0059        // The view delegates the drawing of individual strokes to
#0060        // CStroke::DrawStroke().
#0061        CTypedPtrList<CObList,CStroke*>& strokeList = pDoc->m_strokeList;
#0062        POSITION pos = strokeList.GetHeadPosition();
#0063        while (pos != NULL)
#0064        {
#0065               CStroke* pStroke = strokeList.GetNext(pos);
#0066               pStroke->DrawStroke(pDC);
#0067        }
#0068  }
#0069
#0070  ///////////////////////////////////////////////////////////////////
#0071  // CScribbleView printing
#0072
#0073  BOOL CScribbleView::OnPreparePrinting(CPrintInfo* pInfo)
#0074  {
#0075        // default preparation
#0076        return DoPreparePrinting(pInfo);

#0077  }
#0078
#0079  void CScribbleView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
#0080  {
#0081        // TODO: add extra initialization before printing
#0082  }
#0083
#0084  void CScribbleView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
#0085  {
#0086        // TODO: add cleanup after printing
#0087  }
#0088
#0089  ///////////////////////////////////////////////////////////////////
#0090  // CScribbleView diagnostics
#0091
#0092  #ifdef _DEBUG
#0093  void CScribbleView::AssertValid() const
#0094  {
#0095        CView::AssertValid();
#0096  }
#0097
#0098  void CScribbleView::Dump(CDumpContext& dc) const
#0099  {
#0100        CView::Dump(dc);

#0101  }
#0102
#0103  CScribbleDoc* CScribbleView::GetDocument() // non-debug version is inline
#0104  {
#0105        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CScribbleDoc)));
#0106        return (CScribbleDoc*)m_pDocument;
#0107  }
#0108  #endif //_DEBUG
#0109
#0110  ///////////////////////////////////////////////////////////////////
#0111  // CScribbleView message handlers
#0112
#0113  void CScribbleView::OnLButtonDown(UINT, CPoint point)
#0114  {
#0115     // Pressing the mouse button in the view window starts a new stroke
#0116
#0117        m_pStrokeCur = GetDocument()->NewStroke();
#0118        // Add first point to the new stroke
#0119        m_pStrokeCur->m_pointArray.Add(point);
#0120
#0121        SetCapture();       // Capture the mouse until button up.
#0122        m_ptPrev = point;   // Serves as the MoveTo() anchor point
#0123                            // for the LineTo() the next point,
#0124                      // as the user drags the mouse.
```

```
#0125
#0126         return;
#0127  }
#0128
#0129  void CScribbleView::OnLButtonUp(UINT, CPoint point)
#0130  {
#0131         // Mouse button up is interesting in the Scribble application
#0132         // only if the user is currently drawing a new stroke by dragging
#0133         // the captured mouse.
#0134
#0135         if (GetCapture() != this)
#0136             return; // If this window (view) didn't capture the mouse,
#0137                     // then the user isn't drawing in this window.
#0138
#0139         CScribbleDoc* pDoc = GetDocument();
#0140
#0141         CClientDC dc(this);
#0142
#0143         CPen* pOldPen = dc.SelectObject(pDoc->GetCurrentPen());
#0144         dc.MoveTo(m_ptPrev);
#0145         dc.LineTo(point);
#0146         dc.SelectObject(pOldPen);
#0147         m_pStrokeCur->m_pointArray.Add(point);
#0148
#0149         ReleaseCapture();  // Release the mouse capture established at
#0150                            // the beginning of the mouse drag.
#0151         return;
```

```
#0152  }
#0153
#0154  void CScribbleView::OnMouseMove(UINT, CPoint point)
#0155  {
#0156         // Mouse movement is interesting in the Scribble application
#0157         // only if the user is currently drawing a new stroke by dragging
#0158         // the captured mouse.
#0159
#0160         if (GetCapture() != this)
#0161             return; // If this window (view) didn't capture the mouse,
#0162                     // then the user isn't drawing in this window.
#0163
#0164         CClientDC dc(this);
#0165         m_pStrokeCur->m_pointArray.Add(point);
#0166
#0167         // Draw a line from the previous detected point in the mouse
#0168         // drag to the current point.
#0169         CPen* pOldPen = dc.SelectObject(GetDocument()->GetCurrentPen());
#0170         dc.MoveTo(m_ptPrev);
#0171         dc.LineTo(point);
#0172         dc.SelectObject(pOldPen);
```

```
#0173         m_ptPrev = point;
#0174         return;
#0175  }
```

View 的重绘动作：GetDocument 和 OnDraw

以下是CScribbleView中与重绘动作有关的成员变量和成员函数。

CScribbleView 的成员变量

- m_pStrokeCur：一个指针，指向目前正在工作的线条。
- m_ptPrev：线条中的前一个工作点。我们将在这个点与目前鼠标按下的点之间画一条直线。虽说理想情况下鼠标轨迹的每一个点都应该被记录下来，但如果鼠标移动太快来不及记录，只好在两点之间拉直线。

CScribbleView 的成员函数

- OnDraw：这是一个虚函数，负责将Document的数据显示出来。改写它是程序员最大的责任之一。
- GetDocument：AppWizard 为我们做出这样的代码，以inline方式定义于表头文件：

```
inline CScribbleDoc* CScribbleView::GetDocument()
{ return (CScribbleDoc*)m_pDocument; }
```

其中m_pDocument是CView的成员变量。我们可以推测，当程序设定好 Document Template 之后，每次Framework 动态产生View 对象，其内的 m_pDocument 已经被Framework 设定指向对应之Document了。

View对象何时被动态产生？答案是当使用者选按【File/Open】或【File/New】。每当产生一个 Document，就会产生一组 Document/View/Frame「三口组」。

- OnPreparePrinting， OnBeginPrinting， OnEndPrinting：这三个CView 虚函数将用来改善印表行为。AppWizard 只是先帮我们做出空函数。第12章才会用到它们。

我们来看看CView之中居最重要地位的OnDraw，面对Scribble Document的数据结构，

将如何进行绘图动作。为了获得数据，OnDraw一开始先以GetDocument取得 Document对象指针；然后以while循环一一取得各线条，再调用 CStroke::DrawStroke 绘图。想象中绘图函数应该放在 View 类之内（绘图不正是View 的责任吗），但是DrawStroke 却否！原因是把线条的数据和绘图动作一并放在CStroke中是最好的包装方式。

```
void CScribbleView::OnDraw(CDC* pDC)
{
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // The view delegates the drawing of individual strokes to
    // CStroke::DrawStroke().
    CTypedPtrList<CObList,CStroke*>& strokeList = pDoc->m_strokeList;
    POSITION pos = strokeList.GetHeadPosition();
    while (pos != NULL)
    {
        CStroke* pStroke = strokeList.GetNext(pos);
        pStroke->DrawStroke(pDC);
    }
}
```

其中用到两个CObList 成员函数：

- GetNext：取得下一个元素。
- GetHeadPosition：传回串列之第一个元素的「位置」。传回来的「位置」是一个类型为POSITION 的数值，这个数值可以被使用于CObList的其它成员函数中，例如GetAt或SetAt。你可以把「位置」想象是串列中用以标示某个节点（node）的指针。当然，它并不真正是指针。

View与使用者的交谈(鼠标消息处理实例)

为了实现「以鼠代笔」的功能，CScribbleView 必须接受并处理三个消息：

```
BEGIN_MESSAGE_MAP(CScribbleView, CView)
ON_WM_LBUTTONDOWN()
ON_WM_LBUTTONUP()
ON_WM_MOUSEMOVE()
...
END_MESSAGE_MAP()
```

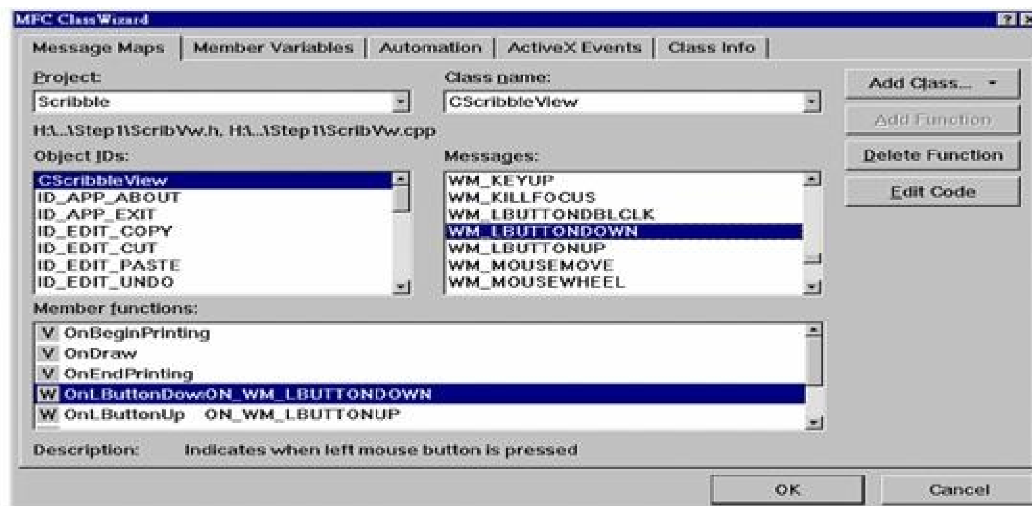
三个消息处理例程的的内容总括来说就是追踪鼠标轨迹、在窗口上绘图、以及调用CStroke成员函数以修正线条内容——包括产生一个新的线条空间以及不断把坐标点加上去。三个函数的重要动作摘记于下。这些函数的骨干及其在 Messape Map中的映射项目，不劳我们动手，有ClassWizard代劳。下一个小节我会介绍其操作方法。

```
void CScribbleView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 当鼠标左键按下，
    // 利用 CScribbleDoc::NewStroke 产生一个新的线条空间；
    // 利用 CArray::Add 把这个点加到线条上去；
    // 调用 SetCapture 取得鼠标捕捉权（mouse capture）；
    // 把这个点记录为 「上一点」（m_ptPrev）；
}
void CScribbleView::OnMouseMove(UINT, CPoint point)
{
    // 当鼠标左键按住并开始移动，
    // 利用 CArray::Add 把新坐标点加到线条上；
    // 在上一点（m_ptPrev）和这一点之间画直线；
    // 把这个点记录为「上一点」（m_ptPrev）；
}
void CScribbleView::OnLButtonUp(UINT, CPoint point)
{
    // 当鼠标左键放开，
    // 在上一点（m_ptPrev）和这一点之间画直线；
    // 利用 CArray::Add 把新的点加到线条上；
    // 调用 ReleaseCapture() 释放鼠标捕捉权（mouse capture）。
}
```

ClassWizard 的辅佐

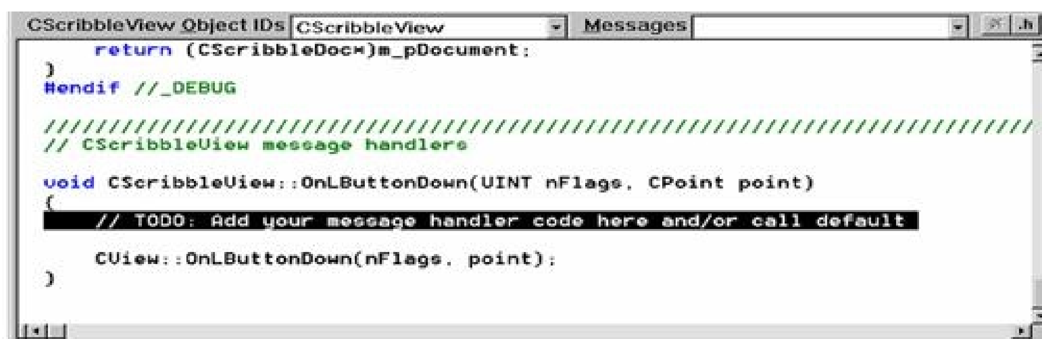
前述三个CScribbleView成员函数（OnLButtonDown，OnLButtonUp，OnMouseMove）是Message Map的一部分，ClassWizard 可以很方便地帮助我们完成相关的Message Map设定工作。

首先，选按【View/ClassWizard】启动ClassWizard，选择其【Message Map】附页：



在图右上侧的【Class Name】清单中选择 CScribbleView，然后在图左侧的【Object IDs】清单中选择 CScribbleView，再在图右侧的【Messages】清单中选择 WM_LBUTTONDOWN，然后选按图右的【Add Function】钮，于是图下侧的【Member functions】清单中出现一笔新项目。

然后，选按【Edit Code】钮，文字编辑器会跳出来，你获得了一个 OnLButtonDown 函数空壳，请在这里键入你的程序代码：



另两个消息处理例程的实现作法雷同。

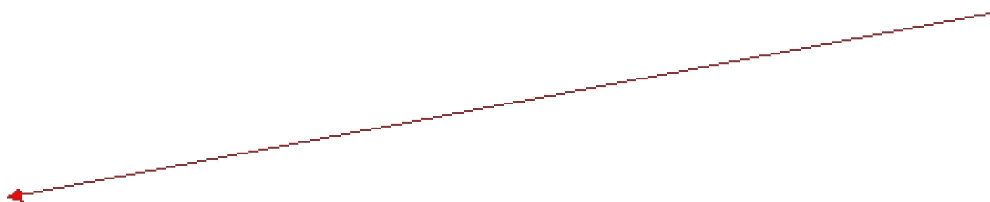
Message Map 因此有什么变化呢？ClassWizard为我们自动加上了三笔映射项目：

```
BEGIN_MESSAGE_MAP(CScribbleView, CView)
        //{{AFX_MSG_MAP(CScribbleView)
        ON_WM_LBUTTONDOWN()
        ON_WM_LBUTTONUP()
        ON_WM_MOUSEMOVE()
        //}}AFX_MSG_MAP
        // Standard printing commands
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

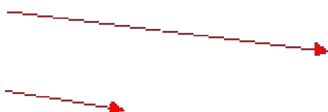
此外ScribbleView的类声明中也自动有了三个成员函数的声明：

```
class CScribbleView : public CView
{
...
// Generated message map functions
protected:
        //{{AFX_MSG(CScribbleView)
        afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
        afx_msg void OnMouseMove(UINT nFlags, CPoint point);
        //}}AFX_MSG
...
};
```

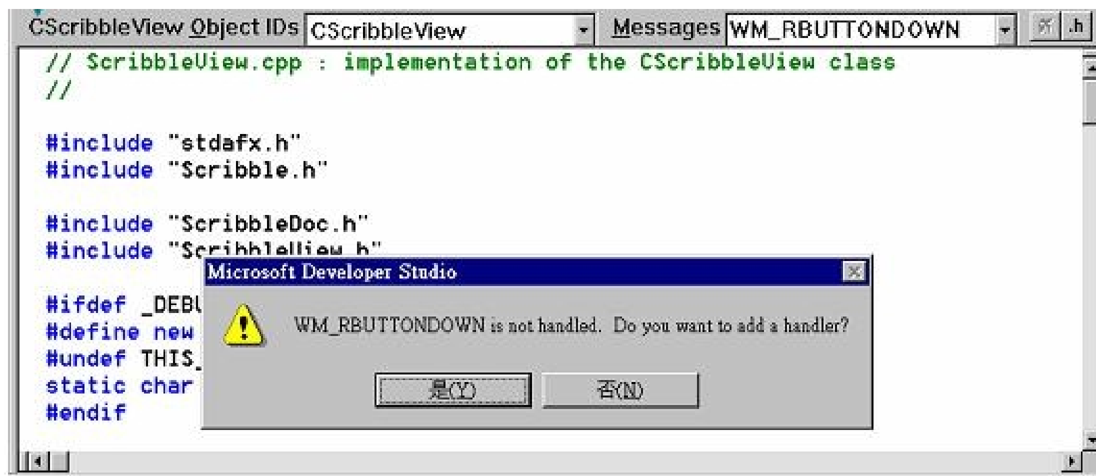
WizardBar的辅佐



WizardBar是Visual C++ 4.0之后的新增工具，也就是文字编辑器上方那个有着【Object IDs】和【Messages】清单的横杆。关于修改Message Map这件事，WizardBar 可以取代ClassWizard 这个大家伙。



首先，进入ScribbleViwe.cpp（因为我们确定要在这里加入三个鼠标消息处理例程），选择WizardBar上的【Object IDs】 为CScribbleView， 再选择【 Messages 】为WM_LBUTTONDOWN，出现以下画面：



回答Yes，于是你获得一个OnLButtonDown 函数空壳，一如在ClassWizard 中所得。请在函数空壳中输入你的程序代码。

Serialize ：对象的文件读写

你可能对Serialization 这个名词感觉陌生，事实上它就是面向对象世界里的 Persistence（永续生存），只是后者比较抽象一些。对象必须能够永续生存，也就是它们必须能够在程序结束时储存到文件中，并且在程序重新启动时再恢复回来。储存和恢复对象的过程在MFC 之中就称为serialization。负责这件重要任务的，是MFC CObject类中一个名为Serialize的虚函数，文件的「读」「写」动作均透过它。

如果文件内容是借着层层类向下管理（一如本例），那么只要每一层把自己份内的工作做好，层层交待下来就可以完成整份数据的文件动作。

Serialization 以外的文件读写动作

其实有时候我们希望在重重包装之中返璞归真一下，感受一些质朴的动作。在介绍 Serialization 的重重包装之前，这里给你一览文件实际读写动作的机会。

文件I/O 服务是任何操作系统的主要服务。Win32 提供了许多文件相关 APIs：开文件、关文件、读文件、写文件、搜寻数据...。MFC 把这些操作都包装在 CFile 之中。可想而知，它必然有 Open、Close、Read、Write、Seek... 等等成员函数。下面这段程序代码示范 CFile 如何读文件：

```
char* pBuffer = new char[0x8000];
CFile file("mydoc.doc", CFile::modeRead); //打开mydoc.doc 文件，使用只读模式。
UINT nBytesRead = file.Read(pBuffer, 0x8000);//读取8000h个字节到 pBuffer中。
```

上述程序片段中，对象file的构造函数将打开mydoc.doc檔。并且由于此对象产生于函数的堆栈之中，当函数结束，file的析构函数将自动关闭mydoc.doc檔。

开文件模式有许多种，都定义在 CFile（AFX.H）之中：

```
enum OpenFlags {
     modeRead =      0x0000,  // 唯讀
     modeWrite =          0x0001,  // 唯寫
     modeReadWrite =0x0002,  // 可讀可寫
     shareCompat =        0x0000,
     shareExclusive =     0x0010,  // 唯我使用
     shareDenyWrite =     0x0020,
     shareDenyRead =0x0030,
     shareDenyNone =0x0040,
     modeNoInherit =0x0080,
     modeCreate =         0x1000,  // 產生新檔（甚至即使已有相同名稱之檔案存在）
     modeNoTruncate =     0x2000,
     typeText =      0x4000, // typeText and typeBinary are used in
     typeBinary =         (int)0x8000 // derived classes only
     };
```

再举一例，下面这段程序代码可将文件mydoc.doc 的所有文字转换为小写：

```
char* pBuffer = new char[0x1000];
CFile file("mydoc.doc", CFile::modeReadWrite);
DWORD dwBytesRemaining = file.GetLength();
UINT nBytesRead;
DWORD dwPosition;

while (dwBytesRemaining) {
    dwPosition = file.GetPosition();
    nBytesRead = file.Read(pBuffer, 0x1000);
    ::CharLowerBuff(pBuffer, nBytesRead);
    file.Seek((LONG)dwPosition, CFile::begin);
    file.Write(pBuffer, nBytesRead);
    dwBytesRemaining -= nBytesRead;
}
delete[] pBuffer;
```

文件的操作常需配合对异常情况（exception）的处理，因为文件的异常情况特别多：檔案找不到啦、文件handles 不足啦、读写失败啦...。上一例加入异样情况处理后如下：

```
char* pBuffer = new char[0x1000];

try {
    CFile file("mydoc.doc", CFile::modeReadWrite);
    DWORD dwBytesRemaining = file.GetLength();
    UINT nBytesRead;
    DWORD dwPosition;

    while (dwBytesRemaining) {
        dwPosition = file.GetPosition();
        nBytesRead = file.Read(pBuffer, 0x1000);
        ::CharLowerBuff(pBuffer, nBytesRead);
        file.Seek((LONG)dwPosition, CFile::begin);
        file.Write(pBuffer, nBytesRead);
        dwBytesRemaining -= nBytesRead;
    }
}
catch (CFileException* e) {
    if (e->cause == CFileException::fileNoteFound)
        MessageBox("File not found");
    else if (e->cause == CFileException::tooManyOpeFiles)
        MessageBox("File handles not enough");
    else if (e->cause == CFileException::hardIO)

        MessageBox("Hardware error");
    else if (e->cause == CFileException::diskFull)
        MessageBox("Disk full");
    else if (e->cause == CFileException::badPath)
        MessageBox("All or part of the path is invalid");
    else
        MessageBox("Unknown file error");
    e->Delete();
}
delete[] pByffer;
```

台面上的 Serialize 动作

让我以Scribble为例，向你解释台面上的（应用程序代码中可见的）serialization 动作。根据图 8-3 的数据结构，Scribble 程序的文件读写动作是这么分工的:

- Framework调用CSribbleDoc::Serialize，用以对付文件。
- CScribbleDoc 再往下调用CStroke::Serialize，用以对付线条。
- CStroke 再往下调用CArray::Serialize，用以对付点数组。

读也由它，写也由它，究竟Serialize 是读还是写？这一点不必我们操心。Framework 调用Serialize 时会传来一个CArchive 对象（稍后我会解释 CArchive），你可以想象它代表一个文件，透过其IsStoring 成员函数，即可知道究竟要读还是写。图8-5 是各层级的Serialize 动作示意图，文字说明已在图片之中。

注意：Scribble 程序使用 `CArray<CPoint, CPoint>` 储存鼠标位置坐标，而CArray 是一个template class，解释起来比较复杂。所以稍后我挖给各位看的 Serialize 函数原始代码，采用CDWordArray 的成员函数而非CArray的成员函数。Visual C++ 1.5版的Scribble 范例程序就是使用 CDWordArray（彼时还未有 template class）。

然而，为求完备，我还是在此先把CArray的Serialize函数原始代码列出：

```
template<class TYPE>
void AFXAPI SerializeElements(CArchive& ar, TYPE* pElements, int nCount)
{
       ASSERT(nCount == 0 ||
              AfxIsValidAddress(pElements, nCount * sizeof(TYPE)));

       // default is bit-wise read/write
       if (ar.IsStoring())
              ar.Write((void*)pElements, nCount * sizeof(TYPE));
       else
              ar.Read((void*)pElements, nCount * sizeof(TYPE));
}

template<class TYPE, class ARG_TYPE>
void CArray<TYPE, ARG_TYPE>::Serialize(CArchive& ar)
{
       ASSERT_VALID(this);
       CObject::Serialize(ar);
       if (ar.IsStoring())
       {
              ar.WriteCount(m_nSize);
       }
       else
       {
              DWORD nOldSize = ar.ReadCount();
              SetSize(nOldSize, -1);
       }
       SerializeElements(ar, m_pData, m_nSize);
}
```

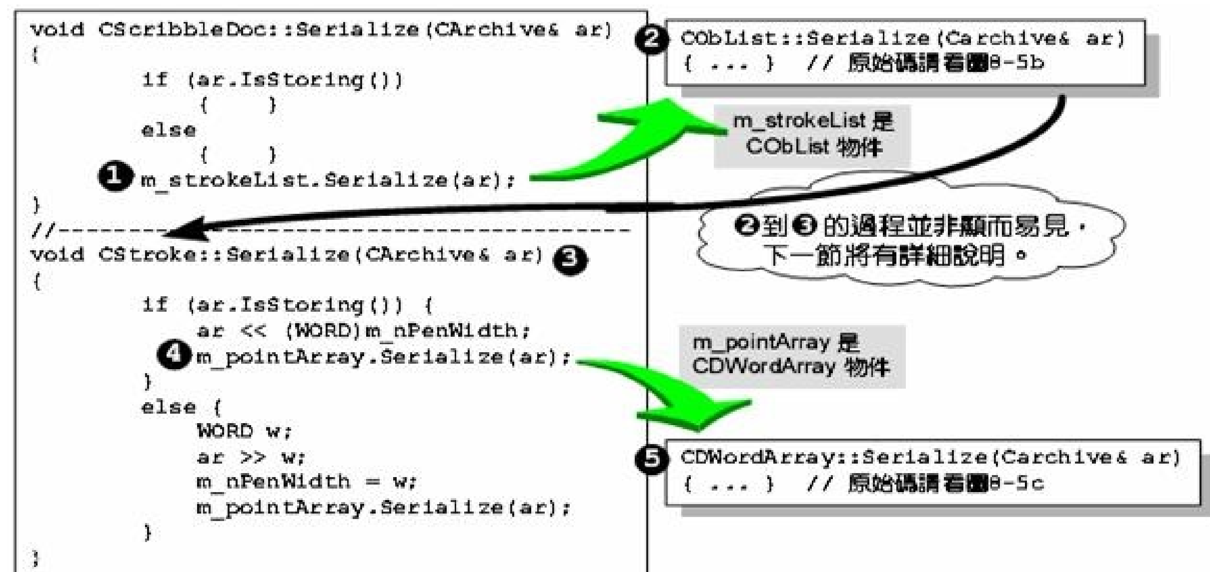


图8-5a Scribble Step1的文件读写（文件）动作

```
void CObList::Serialize(CArchive& ar)
{
    ASSERT_VALID(this);
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        ar.WriteCount(m_nCount);
        for (CNode* pNode = m_pNodeHead; pNode != NULL;
             pNode = pNode->pNext)
        {    // J.J.Hou : 針對串列中的每一個元素寫檔
            ASSERT(AfxIsValidAddress(pNode, sizeof(CNode)));
            ar << pNode->data;
        }
    }
    else
    {
        DWORD nNewCount = ar.ReadCount();
        CObject* newData;
        while (nNewCount--)
        {   // J.J.Hou : ——讀入檔案內容，加入串列
            ar >> newData;
            AddTail(newData);
        }
    }
}
```


图8-5b CObList::Serialize 原始代码

```
void CDWordArray::Serialize(CArchive& ar)
{
    ASSERT_VALID(this);
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        ar.WriteCount(m_nSize);              // 把陣列大小（元素個數）寫入 ar
        ar.Write(m_pData, m_nSize * sizeof(DWORD));  // 把整個陣列寫入 ar
    }
    else
    {
        DWORD nOldSize = ar.ReadCount();
        SetSize(nOldSize);                   // 從 ar 中讀出陣列大小（元素個數）
        ar.Read(m_pData, m_nSize * sizeof(DWORD));   // 從 ar 中讀出整個陣列
    }
}
```

图8-5c CDWordArray::Serialize 原始代码

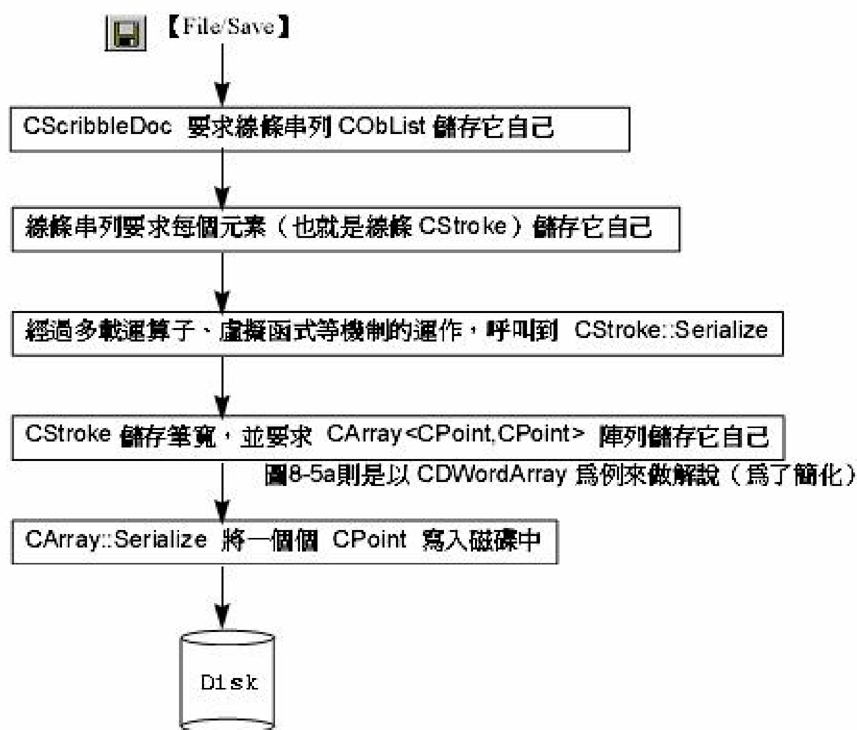


图8-5d Scribble Document的Serialize动作细部分解

实际看看储存在磁盘中的 .SCB 文件内容，对Serialize 将会有深刻的体会。图8-6a 是使用者在Scribble Step1程序的绘图画面及存盘内容（以Turbo Dump观察获得），图 8-6b 是文件内容的解释。我们必须了解隐藏在 MFC 机制中的serialization细部动作，才能清楚这些二进位数据的产生原由。如果你认为看倾印代码（dump code）是件令人头晕的事情，那么你会错失许多美丽事物。真的，倾印代码使我们了解许多深层结构。

我在Scribble 中作画并存文件。为了突显笔宽的不同，我用了第10章的 Step3 版本，该版本的 Document 格式与 Step1 的相同，但允许使用者设定笔宽。图 8-6a 第一条线条的笔宽是 2，第二条是 5，第三条是 10，第四条是 20。文件储存于 PENWIDTH.SCB 文件中。

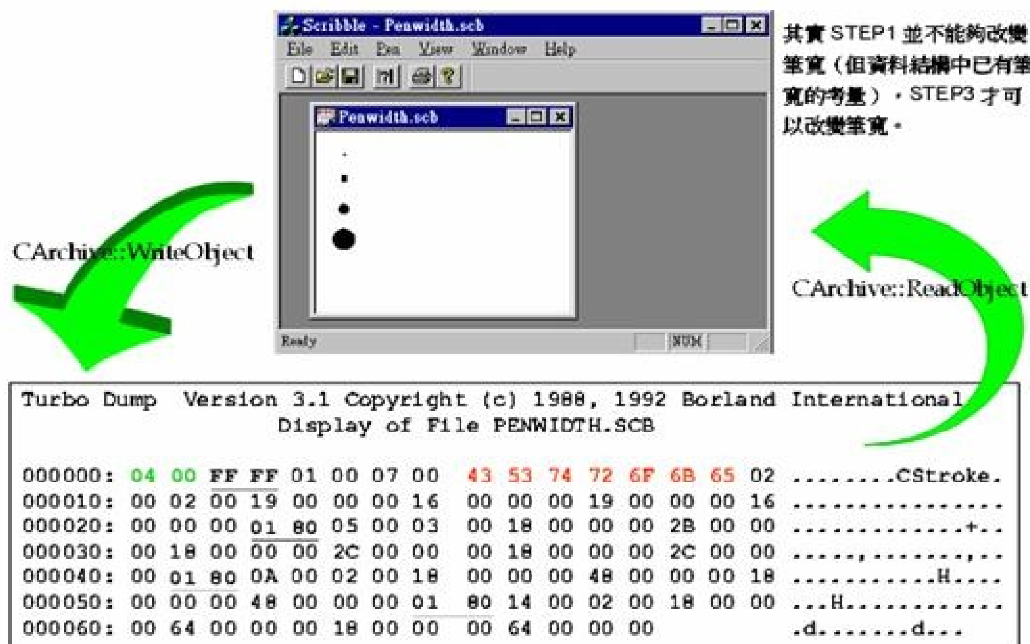


图8-6a 在Scribble中作画并存文件。PENWIDTH.SCB 文件全长109 个字节

數值（hex）	說明
0004	表示此檔有四個 *CObList* 元素。
FFFF	FFFF 亦即 -1，表示 New Class Tag（稍後詳述）。既然是新類別，就得記錄一些相關資訊（版本號碼和類別名稱）
0001	這是 Schema no.，代表物件的版本號碼。此數值由 *IMPLEMENT_SERIAL* 巨集的第三個參數指定。
0007	表示後面接著的「類別名稱」有 7 個字元。
43 53 74 72 6F 6B 65	"CStroke"（類別名稱）的 ASCII 碼。
0002	第一條線條的寬度。
0002	第一條線條的點陣列大小（點數）。
00000019,00000016	第一條線條的第一個點座標（*CPoint* 物件）。
00000019,00000016	第一條線條的第二個點座標（*CPoint* 物件）。
8001	這是（*wOldClassTag* | *nClassIndex*）的組合結果，表示接下來的物件仍舊使用舊類別（稍後詳述）
0005	第二條線條的寬度。
0003	第二條線條的點陣列大小（點數）。
00000018,0000002B	第二條線條的第一個點座標（*CPoint* 物件）。
00000018,0000002C	第二條線條的第二個點座標（*CPoint* 物件）。
00000018,0000002C	第二條線條的第三個點座標（*CPoint* 物件）。
8001	表示接下來的物件仍舊使用舊類別。
000A	第三條線條的寬度。
0002	第三條線條的點陣列大小（點數）。
0000018,00000048	第三條線條的第一個點座標（*CPoint* 物件）。
0000018,00000048	第三條線條的第二個點座標（*CPoint* 物件）。
8001	表示接下來的物件仍舊使用舊類別。
0014	第四條線條的寬度。
0002	第四條線條的點陣列大小（點數）。
0000018,00000064	第四條線條的第一個點座標（*CPoint* 物件）。
0000018,00000064	第四條線條的第二個點座標（*CPoint* 物件）。

图8-6b PENWIDTH.SCB 文件内容剖析。别忘了Intel采用 "little-endian" 字节排列方式，每一个字组的前后字节系颠倒放置

台面下的 **Serialize** 写文件奥秘

你属于打破砂锅问到底，不到黄河心不死那一型吗？我会满足你的好奇心。从应用程序代码的层面来看，关于文件的读写，我们有许多环节无法打通，类的层层调用动作似乎有几个缺口，而图8-6a文件文件倾印代码中神秘的 `FF FF 01 00 07 00 43 53 74 72 6F 6B 65` 也暧昧难

明。现在让我来抽丝剥茧。

在挖宝过程之中，我们当然需要一些工具。我不选用昂贵的电钻、空压机或怪手（因为你可能没有），我只选用简单的鹤嘴锄和铲子：一个文字搜寻工具，一个文件倾印工具，一个Visual C++ 内含的除错器。

- GREP.COM：UNIX上赫赫有名的文字搜寻工具，Borland C++ 编译器套件附了一个DOS版。此工具可以为我们搜寻文件中是否有特定字符串。PC Tools 也有这种功能，但PC Tools属于重量级装备，不符合我的选角要求。GREP的使用方式如下：

```
E:\MSDEV\MFC\SRC> grep -d Serialize *.cpp <Enter>
```

搜尋對象

欲搜尋之字串（如果中有空白，可用雙引號整個括起來）

-d 表示子目錄一併搜尋（此爲選項）

- TDUMP.EXE：Turbo Dump，Borland C++ 所附工具，可将任何文件以16进位代码显示。使用方式如下：

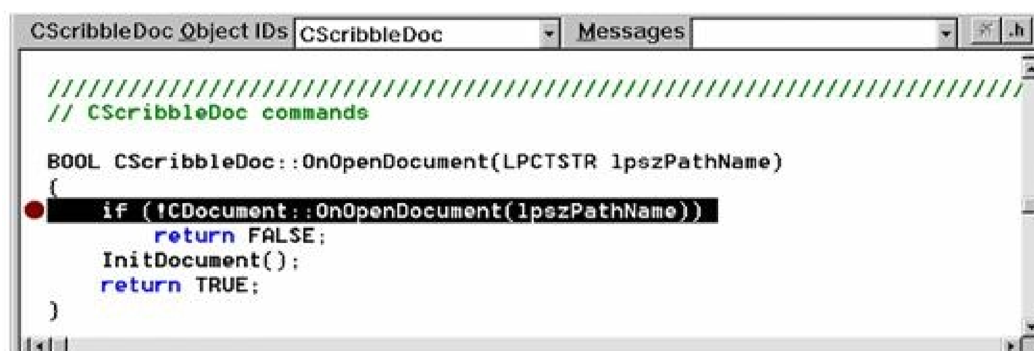
```
C:\> tdump penwidth.scb   （输出结果将送往屏幕）
```

或

```
C:\> tdump penwidth.scb > filename   （输出结果将送往文件）
```

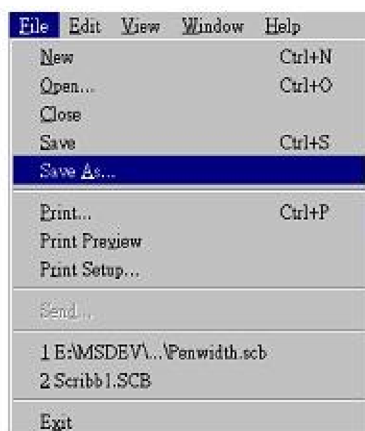
- Visual C++ 除错器：我已在第 4 章介绍过这个除错器。我假设你已经懂得如何设定中断点、观察变量值，并以Go、Step Into、Step Over、Step Out、Step to Cursor 进行除错。这里我要补充的是如何观察"Call Stack"。

如果我把中断点设在 CScribbleDoc::OnOpenDocument 函数中的第一行，



然后以Go进入除错程序，当我在Scribble 中打开一份文件（首先面对一个对话框，然后指定文件名），程序停留在中断点上，然后我选按【View/Call Stack】，出现【Call Stack】窗口，把中断点之前所有未结束的函数列出来。这份数据可以帮助我们挖掘MFC。

好，图 8-5a的函数流程使图8-6a 的文件文件倾印代码曙光乍现，但是其中有些关节仍还模模糊糊，旋明旋暗。那完全是因为 CObList 在处理每一个元素（一个 CObject 派生类之对象实体）的文件动作时，有许多幕后的、不易观察到的机制。让我们从使用者按下【Save As】选单项目开始，追踪程序的进行。



遍尋 Scribble 程式，並沒有發現曾經在哪裡攔截過【Save As】命令訊息，那麼必是某個「CCmdTarget 衍生類別」曾經在其 Message Map 中設定過對此訊息之處理函式。我猜想 CDocument 最有這個可能：

```
BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
        //{{AFX_MSG_MAP(CDocument)
        ON_COMMAND(ID_FILE_CLOSE, OnFileClose)
        ON_COMMAND(ID_FILE_SAVE, OnFileSave)
        ON_COMMAND(ID_FILE_SAVE_AS, OnFileSaveAs)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

賓果！於是【Save As】引發 CDocument::OnFileSaveAs 被呼叫。

```
void CDocument::OnFileSaveAs()
{
    if (!DoSave(NULL))
        TRACE0("Warning: File save-as failed.\n");
}
```




```
BOOL CDocument::DoSave(LPCTSTR lpszPathN
{
    CString newName = lpszPathName;
    if (newName.IsEmpty())
    {
        CDocTemplate* pTemplate = GetDoc
        newName = m_strPathName;
        ...

        if (!AfxGetApp()->DoPromptFileName(newName,
          bReplace ? AFX_IDS_SAVEFILE : AFX_IDS_SAVEFILECOPY,
          OFN_HIDEREADONLY | OFN_PATHMUSTEXIST, FALSE, pTemplate))
            return FALSE;    // don't even attempt to save
    }

    CWaitCursor wait;

    if (!OnSaveDocument(newName))
    { ... }

    ...
}
```

下頁

```
BOOL CDocument::OnSaveDocument(LPCTSTR lpszPathName)
{
    CFileException fe;
    CFile* pFile = NULL;
    pFile = GetFile(lpszPathName, CFile::modeCreate |
            CFile::modeReadWrite | CFile::shareExclusive, &fe);

    CArchive saveArchive(pFile, CArchive::store |
                        CArchive::bNoFlushOnDelete);
    saveArchive.m_pDocument = this;
    saveArchive.m_bForceFlat = FALSE;
    TRY
    {
        CWaitCursor wait;
        Serialize(saveArchive);
        saveArchive.Close();
        ReleaseFile(pFile, FALSE);
    }
    ...
}
```

雖然看起來像是呼叫 CDocument::Serialize，但事實上因爲 Serialize 是虛擬函式，而 CScribbleDoc 已改寫它，而且目前的 this 指標是指向 CScribbleDoc 物件（別忘了整個追蹤路線的起源是 Scribble Docment；我會在下一章訊息繞行這一主題中解釋更詳盡一些），所以這裡呼叫的是 CScribbleDoc::Serialize 函式。哈，這就是虛擬函式的妙用！

```
void CScribbleDoc::Serialize(CArchive& ar)
{
    ...
    m_strokeList.Serialize(ar);
}
```

m_strokeList 是個 CObList 物件

```
void CObList::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        ar.WriteCount(m_nCount);
        for (CNode* pNode = m_pNodeHead;
             pNode != NULL; pNode = pNode->pNext)
        {
            ar << pNode->data;
        }
    }
    else
    {
        ...
    }
}
```

本例之 CObList 物件內有 4 個元素，所以輸出資料 0004

for loop

CArchive 已針對 << 運算子做了多載（overloading）動作。

```
_AFX_INLINE CArchive& AFXAPI operator<<(CArchive& ar,
                                        const CObject* pOb)
        { ar.WriteObject(pOb); return ar; }
```

呼叫 CArchive::WriteObject

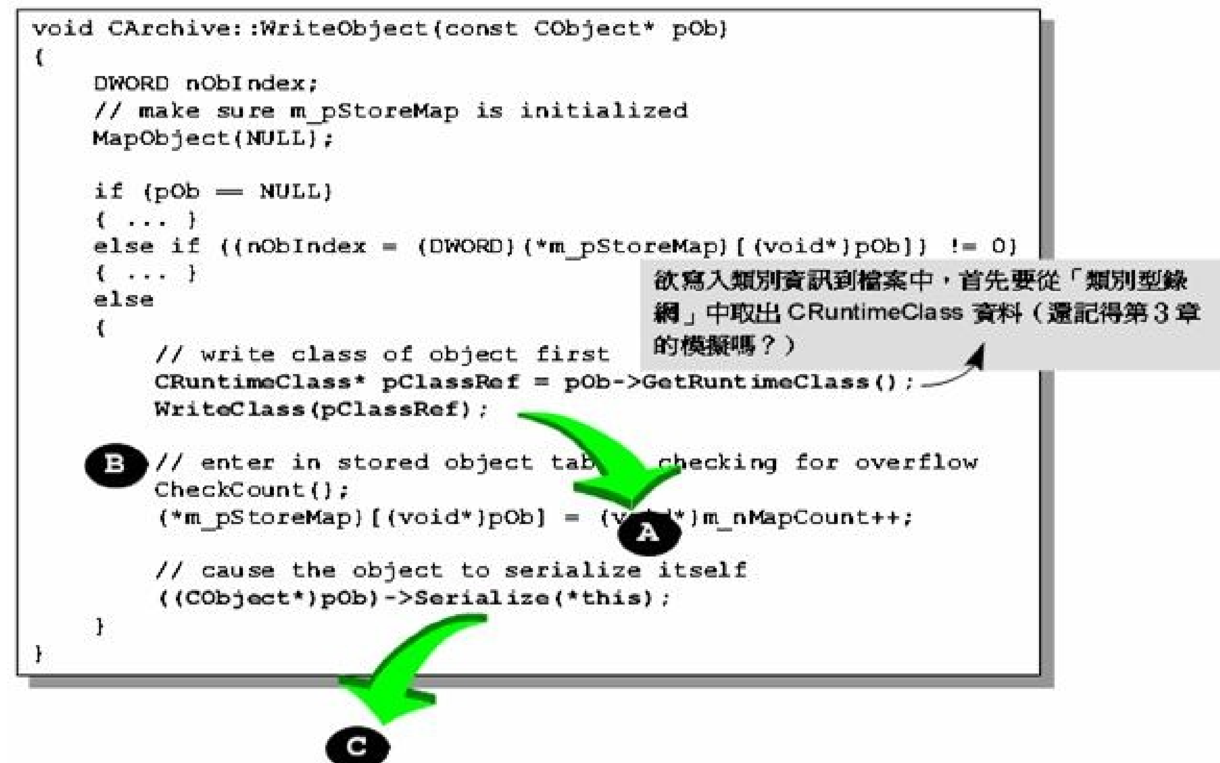
下頁

你属于打破砂锅问到底，不到黄河心不死那一型吗？这段刨根究底的过程应能解你疑惑。根据我的经验，经过这么一次巡礼，我们就能够透析 MFC 的内部运作并确实掌握MFC的类运用了。换言之，我们现在到达知其所以然的境界了。

台面下的 **Serialize** 读文件奥秘

大大地喘口气吧，能够把MFC的Serialize 写文件动作完全摸透，是件值得慰劳自己的「功绩」。但是你只能轻松一下下，因为读文件动作还没有讨论过，而读文件绝不只是「写文件的逆向操作」而已。

把对象从文件中读进来，究竟技术关键在哪里？读取数据当然没问题，问题是「Document/View/Frame 三口组」怎么产生？从文件中读进一个类名称，又如何动态产生其对象？当我从文件读到"CStroke" 这个字符串，并且知道它代表一个类名称，然后我怎么办？我能够这么做吗：



```
#define wNullTag        ((WORD)0)
#define wNewClassTag    ((WORD)0xFFFF)
#define wClassTag       ((WORD)0x8000)
#define dwBigClassTag   ((DWORD)0x80000000)
#define wBigObjectTag   ((WORD)0x7FFF)
#define nMaxMapCount    ((DWORD)0x3FFFFFFE)

void CArchive::WriteClass(const CRuntimeClass* pClassRef)
{
    if (pClassRef->m_wSchema == 0xFFFF)
    {
      TRACE1("Warning: Cannot call WriteClass/WriteObject for %hs.\n",
              pClassRef->m_lpszClassName);
      AfxThrowNotSupportedEx
    }
    ...
    DWORD nClassIndex;
    if ((nClassIndex = (DWORD)(*m_pStoreMap)[(void*)pClassRef]) != 0)
    {
      // previously seen class, write out the index tagged by high bit
      if (nClassIndex < wBigObjectTag)
          *this << (WORD)(wClassTag | nClassIndex);
      else
      {
          *this << wBigObjectTag;
          *this << (dwBigClassTag | nClassIndex);
      }
    }
    else
    {
        // store new class
        *this << wNewClassTag;
        pClassRef->Store(*this);
        ...
    }
}
```

A

本例 CObList 串列內之元素種類（也就是「CObject 衍生類別」）只有一種（CStroke），所以 nClassIndex 永遠爲 1。於是輸出資料 8001。

遇到串列中的新元素（新類別），就輸出資料 FFFF。

下頁

```
CString aStr;
... // read a string from file to aStr
CStroke* pStroke = new aStr;
```

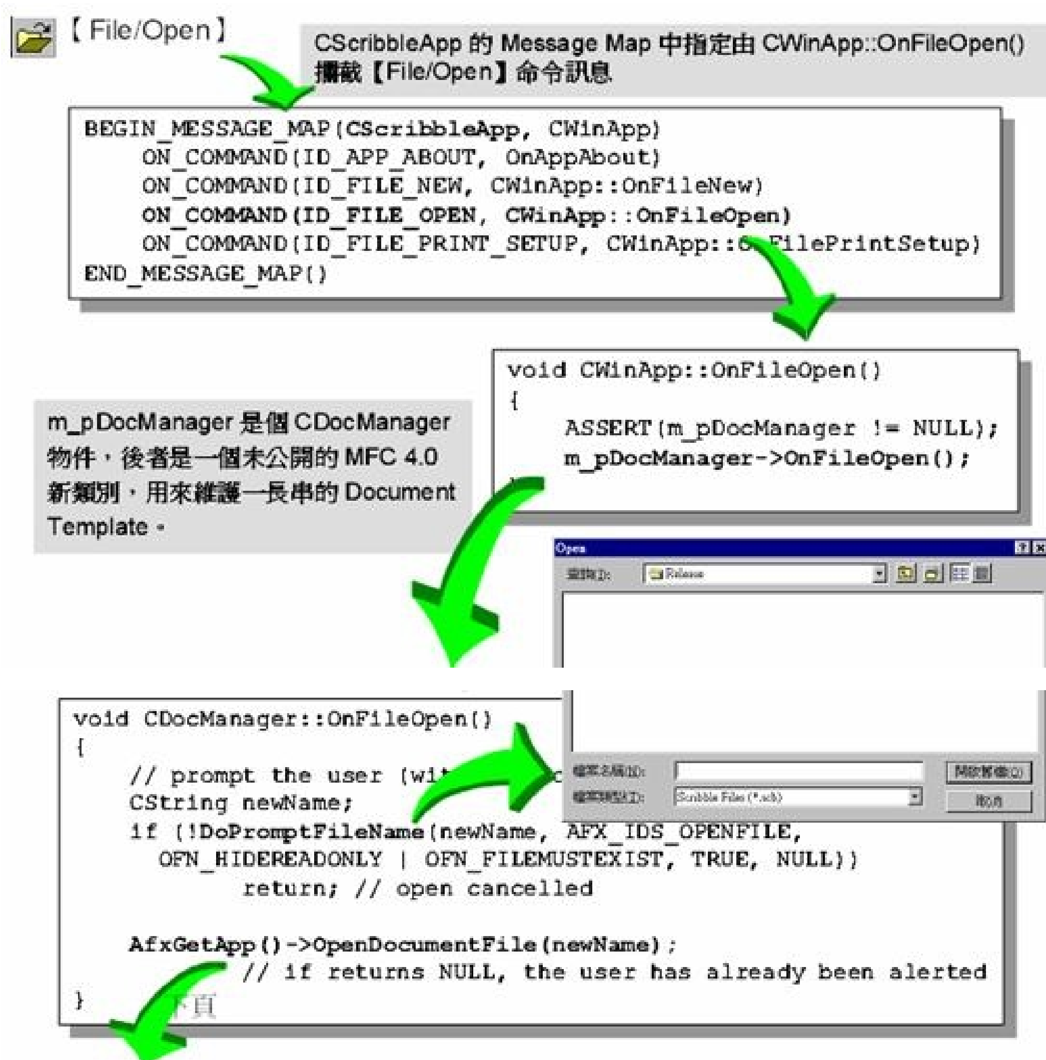
不行！这是语言版的动态生成；没有任何一个C++编译器支持这种能力。那么我能够这么做吗：

```
CString aStr;
CStroke* pStroke;
... // read a string from file to aStr
if (aStr == CString("CStroke")
   CStroke* pStroke = new CStroke;
else if (aStr == CSting("C1")
   C1* pC1 = new C1;
else if (aStr == CSting("C2")
   C2* pC2 = new C2;
else if (aStr == CSting("C3")
   C1* pC3 = new C3;
else ...
```

可以，但真是粗糙啊。万一再加上一种新类呢？万一又加上一种新类呢？不胜其扰也！

第 3 章已经提出动态生成的观念以及实现方式了。主要关键还在于一个「类型录网」。这个型录网就是 CRuntimeClass 组成的一个串列。每一个想要享有动态生成机能的类，都应该在「类型录网」上登记有案，登记数据包括对象的构造函数的指针。也就是说，上述那种极不优雅的比对动作，被 MFC 巧妙地埋起来了；应用程序可以风姿优雅地，单单使用 DECLARE_SERIAL和 IMPLEMENT_SERIAL两个宏，就获得文件读写以及动态生成两种机制。

我将仿效前面对于写文件动作的探索，看看读文件的程序如何。



```
CDocument* CWinApp::OpenDocumentFile(LPCTSTR lpszFileName)
{
    ASSERT(m_pDocManager != NULL);
    return m_pDocManager->OpenDocumentFile(lpszFileName);
}
```

很多原先在 CWinApp 中做掉的有關於 Document Template 的工作，如 AddDocTemplate、OpenDocumentFile 和 NewDocumentFile，自從 MFC 4.0 之後已隔離出來由 CDocManager 負責。

```
CDocument* CDocManager::OpenDocumentFile(LPCTSTR lpszFileName)
{
    // find the highest confidence
    CDocTemplate* pBestTemplate = NULL;
    CDocument* pOpenDocument = NULL;
    TCHAR szPath[_MAX_PATH];
    ... // 從「Document Template 串列」中找出最適當之 template，
    ... // 放到 pBestTemplate 中。
    return pBestTemplate->OpenDocumentFile(szPath);
}
```

由於 CMultiDocTemplate 改寫了 OpenDocumentFile，所以呼叫的是CMultiDocTemplate::OpenDocumentFile。

下頁

```
CDocument* CMultiDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName,
            BOOL bMakeVisible)
{
    CDocument* pDocument = CreateNewDocument();
    ...
    CFrameWnd* pFrame = CreateNewFrame(pDocument, NULL);
    ...

    if (lpszPathName == NULL)
    {
        // create a new document - with default document name
        ...
    }
    else
    {
        // open an existing document
        CWaitCursor wait;
        if (!pDocument->OnOpenDocument(lpszPathName))
        {
            ...
        }
        pDocum...
    }
    InitialUpdateFrame(pFrame, pDocument, bMakeVisible);
    return pDocument;
}
```

原始碼請見本章前部之 "CDocTemplate管理 CDocument/CView/CFrameWnd" 一節。

由於 CScribbleDoc 改寫了 OnOpenDocument，所以呼叫的是CScribbleDoc::OnOpenDocument

```
BOOL CScribbleDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;
    InitDocument();
    return TRUE;
}
```

下頁

```
BOOL CDocument::OnOpenDocument(LPCTSTR lpszPathName)
{
    CFileException fe;
    CFile* pFile = GetFile(lpszPathName,
            CFile::modeRead|CFile::shareDenyWrite, &fe);

    DeleteContents();
    SetModifiedFlag();  // dirty during de-serialize

    CArchive loadArchive(pFile, CArchive::load |
                         CArchive::bNoFlushOnDelete);
    loadArchive.m_pDocument = this;
    loadArchive.m_bForceFlat = FALSE;
    TRY
    {
        CWaitCursor wait;
        if (pFile->GetLength() != 0)
            Serialize(loadArchive);      // load me
        loadArchive.Close();
        ReleaseFile(pFile, FALSE);
    }
    ...
}
```

由於 CScribbleDoc 改寫了 Serialize，
所以呼叫的是 CScribbleDoc::Serialize

```
void CScribbleDoc::Serialize(CArchive& ar)
{
    ...
    m_strokeList.Serialize(ar);
}
```

```
void CObList::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        ...
    }
    else
    {
        DWORD nNewCount = ar.ReadCount();
        CObject* newData;
        while (nNewCount--)
        {
            ar >> newData;
            AddTail(newData);
        }
    }
}
```

本例讀入 0004

for loop

operator>> 被多載（overloading）化

```
_AFX_INLINE CArchive& AFXAPI operator>>(CArchive& ar,
                                        CObject*& pOb)
        { pOb = ar.ReadObject(NULL); return ar; }
```

呼叫 CArchive::ReadObject

下頁

```
CObject* CArchive::ReadObject(const CRuntimeClass*
pClassRefRequested)
{
    ...
    // attempt to load next stream as CRuntimeClass
    UINT nSchema;
    DWORD obTag;
    CRuntimeClass* pClassRef = ReadClass(pClassRefRequested,
                                         &nSchema, &obTag);

    // check to see if tag to already loaded object
    CObject* pOb;
    if (pClassRef == NULL)
    {  ...  }
    else
    {
        // allocate a new object based on the c
        pOb = pClassRef->CreateObject();

        // Add to mapping array BEFORE de-s
        CheckCount();
        m_pLoadArray->InsertAt(

        // Serialize the object
        UINT nSchemaSave = m_nOb
        m_nObjectSchema = nSchema;
        pOb->Serialize(*this);
        m_nObjectSchema = nSchemaSave;
    }

    return pOb;
}
```

IMPLEMENT_SERIAL(CStroke,...)
會展開出一個函式如下，此即動態生成的奧秘：

```
CObject* PASCAL CStroke::CreateObject()
{
   return new CStroke;
}
```

呼叫 CStroke::Serialize

A　B　C　D

```
CRuntimeClass* CArchive::ReadClass(const CRuntimeClass* pClassRefRequested,
        UINT* pSchema, DWORD* pObTag)
{
    WORD wTag;
    *this >> wTag;
    ...
    CRuntimeClass* pClassRef;
    UINT nSchema;
    if (wTag == wNewClassTag)
    {
        // new object follows a new class id
        if ((pClassRef = CRuntimeClass::Load(*this, &nSchema)) == NULL)
            ...
    }
    else
    {
        DWORD nClassIndex;
        // 判斷 nClassIndex 爲舊類別，於是從類別型錄網中取出
        // 其 CRuntimeClass，放在 pClassRef 中。
        ...
    }
    ...
    return pClassRef;
}
```

本例讀入 FFFF

B

```
CRuntimeClass* PASCAL CRuntimeClass::Load(CArchive& ar,
        UINT* pwSchemaNum) // loads a runtime class description
{
    WORD nLen;
    char szClassName[64];
    CRuntimeClass* pClass;

    WORD wTemp;
    ar >> wTemp;
    *pwSchemaNum
    ar >> nLen;

    if (nLen >= _countof(szClassName) ||
        ar.Read(szClassName, nLen*sizeof(char)) != nLen*sizeof(char))
    {
            return NULL;
    }
    szClassName[nLen] = '\0';

    // search app specific classes
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    AfxLockGlobals(CRIT_RUNTIMECLASSLIST);
    for (pClass = pModuleState->m_classList; pClass != NULL;
         pClass = pClass->m_pNextClass)
    {
            if (lstrcmpA(szClassName, pClass->m_lpszClassName) == 0)
            {
                    AfxUnlockGlobals(CRIT_RUNTIMECLASSLIST);
                    return pClass;
            }
    }
    ...
}
```


D

```
void CStroke::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ...
    }
    else
    {
        WORD w;
        ar >> w;
        m_nPenWidth = w;
        m_pointArray.Serialize(ar);
    }
}
```



```
void CDWordArray::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);

    if (ar.IsStoring())
    {
        ...
    }
    else
    {
        DWORD nOldSize = ar.ReadCount();
        SetSize(nOldSize);
        ar.Read(m_pData, m_nSize * sizeof(DWORD));
    }
}
```

DYNAMIC / DYNCREATE / SERIAL 三宏

我猜你被三组看起来难分难解的宏困扰着，它们是：

- DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC
- DECLARE_DYNCREATE / IMPLEMENT_DYNCREATE
- DECLARE_SERIAL / IMPLEMENT_SERIAL

事实上我已经在第 3 章揭露其原始代码及其观念了。这里再以图 8-7 三张图片把宏原始代码、展开结果、以及带来的影响做个整理。SERIAL宏中比较令人费解的是它对 `>>` 运算符的重载动作。稍后我有一个CArchive小节，会交待其中细节。

你将在图8-7abc中看到几个令人困惑的大写常数，像是AFXAPI、AFXDATA等等。它们的意义可以在 VC++ 5.0 的\DEVSTUDIO\VC\MFC\INCLUDE\AFXVER_.H 中获得：

```
// AFXAPI is used on global public functions
#ifndef AFXAPI
        #define AFXAPI __stdcall
#endif

#define AFX_DATA
#define AFX_DATADEF
```

后二者就像afx_msg一样（我曾经在第 6 章的 Hello MFC 原始代码一出现之后解释过），是一个"intentional placeholder"，可能在将来会用到，目前则为「无物」。

DYNAMIC / DYNCREATE / SERIAL三套宏分别在CRuntimeClass 所组成的「类型录网」中填写不同的记录，使MFC 类（以及你自己的类）分别具备三个等级的性能：

- 基础机能以及对象诊断（可利用afxDump 输出诊断消息），以及Run Time Type Information（RTTI）。也有人把RTTI称为Run Time Class Information（RTCI）。
- 动态生成（Dynamic Creation）
- 文件读写（Serialization）

你的类究竟拥有什么等级的性能，得视其所使用的宏而定。三组宏分别实现不同等级的功能，如图 8-8。

功能 巨集	RTTI CObject::IsKindOf	Dynamic Creation CRuntimeClass::CreateObject	Serialize CArchive::operator>> CArchive::operator<<
DYNAMIC	Yes	No	No
DYNCREATE	Yes	Yes	No
SERIAL	Yes	Yes	Yes

Scribble Step1程序中与主结构相关的六个类，所使用的各式宏整理如下：

类名称	基类	使用之宏
CScribbleApp	CWinApp	None
CMainFrame	CMDIFrameWnd	DECLARE_DYNAMIC(CMainFrame) IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)
CChildFrame	CMDIChildWnd	DECLARE_DYNCREATE(CChildFrame) IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)
CScribbleDoc	CDocument	DECLARE_DYNCREATE(CScribbleDoc) IMPLEMENT_DYNCREATE(CscribbleDoc, CDocument)
CStroke	CObject	DECLARE_SERIAL(CStroke) IMPLEMENT_SERIAL(Cstroke, Cobject, 1)
CScribbleView	CView	DECLARE_DYNCREATE(CScribbleView) IMPLEMENT_DYNCREATE(CscribbleView, CView)

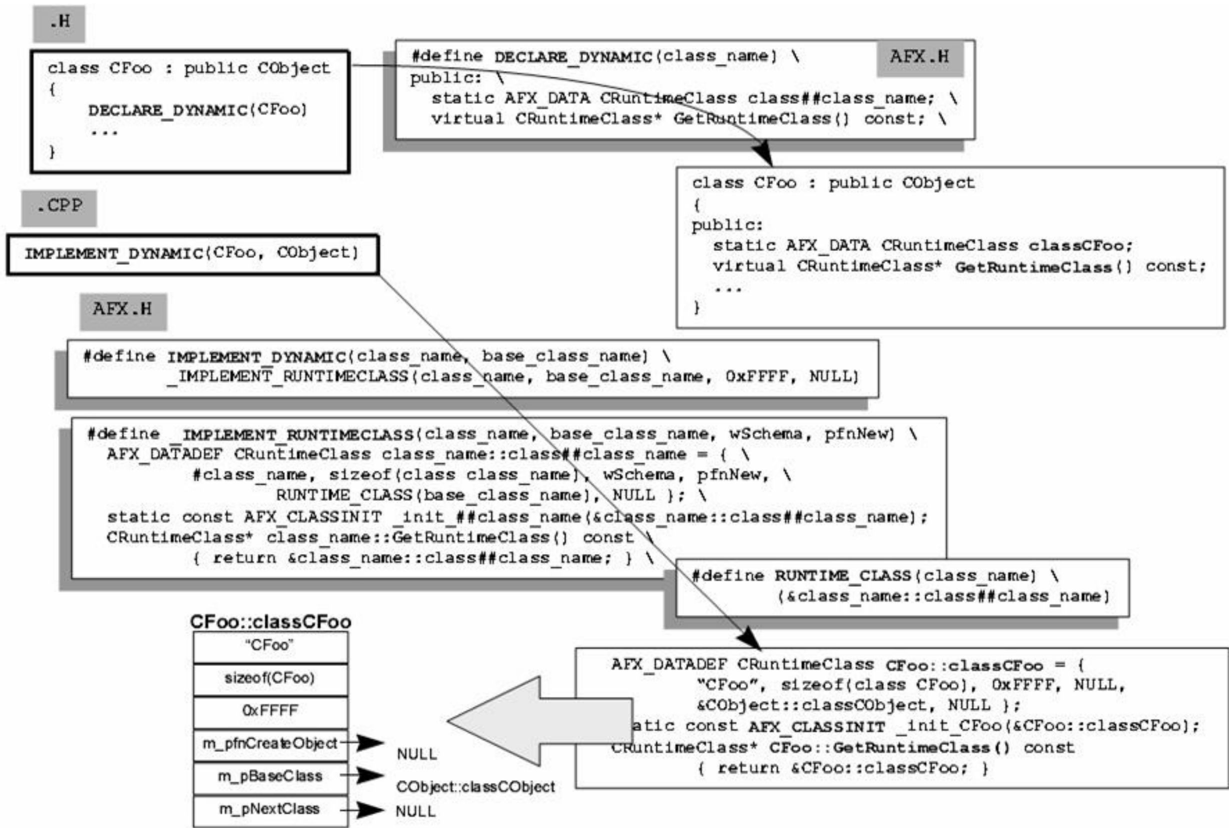


圖8-7a DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC

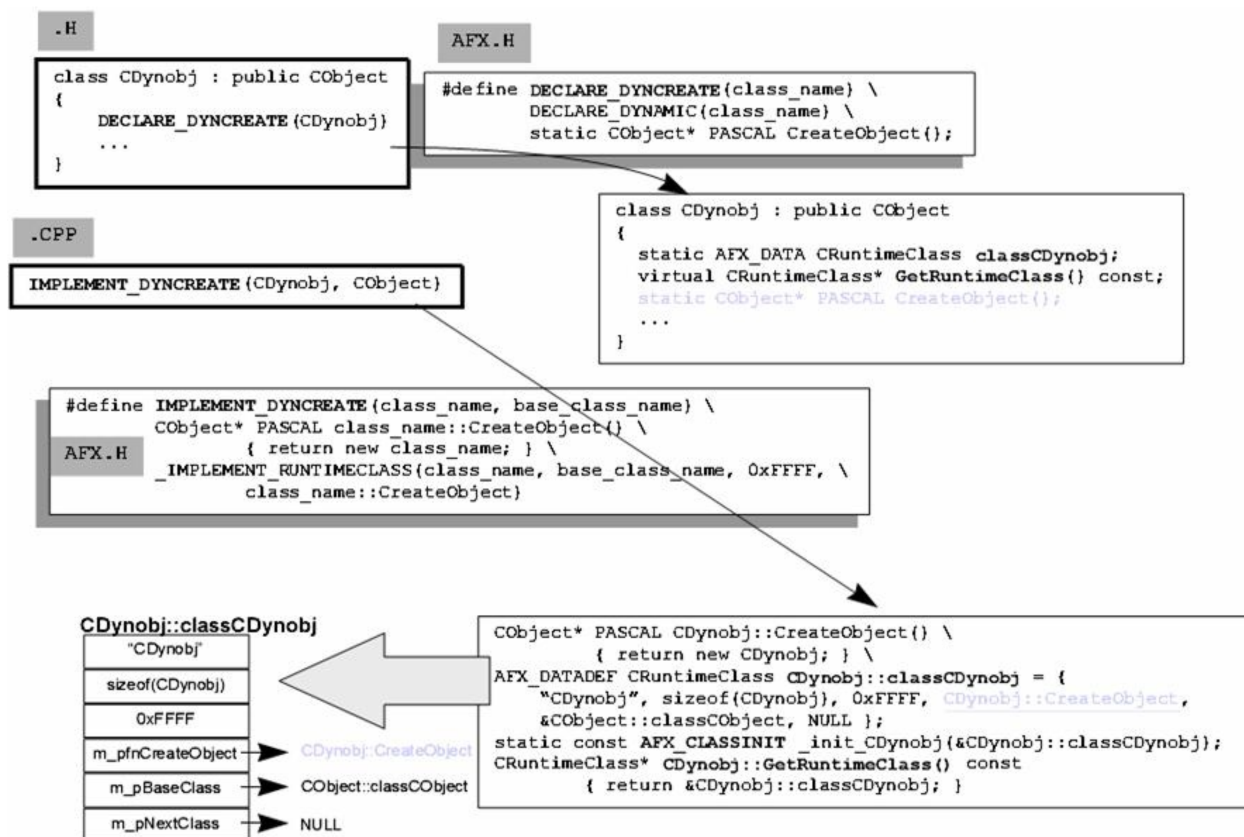


图8-7b DECLARE_DYNCREATE / IMPLEMENT_DYNCREATE

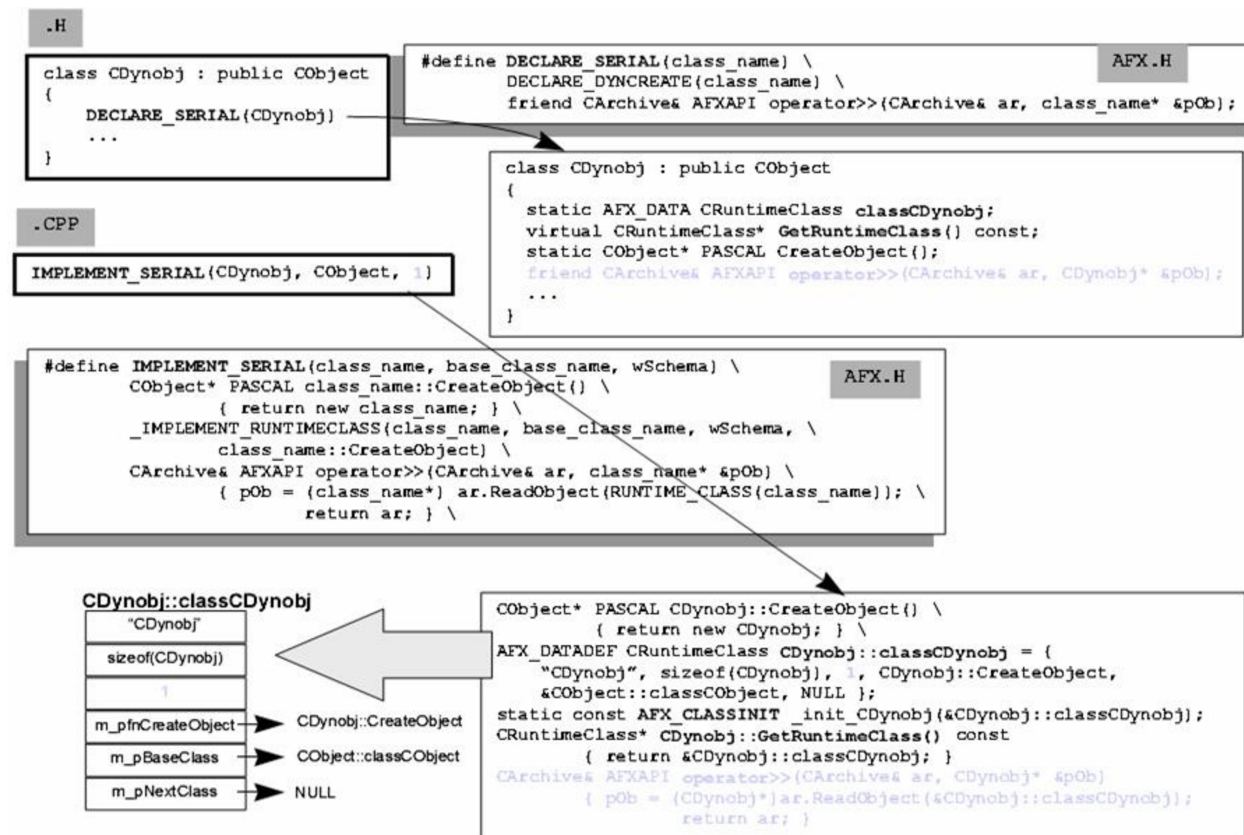


图8-7c DECLARE_SERIAL / IMPLEMENT_SERIAL

Serializable 的必要条件

欲让一个对象有Serialize 能力，它必须派生自一个Serializable 类。一个类意欲成为Serializable，必须有下列五大条件；至于其原因，前面的讨论已经全部交待过了。

1. 从CObject 派生下来。如此一来可保有RTTI、Dynamic Creation 等机能。

2. 类的声明部分必须有DECLARE_SERIAL 宏。此宏需要一个参数：类名称。

3. 类的实现部分必须有IMPLEMENT_SERIAL宏。此宏需要三个参数：一是类名称，二是父类名称，三是schema no.。

4. 改写Serialize 虚函数，使它能够适当地把类的成员变量写入文件中。

5. 为此类加上一个default 构造函数（也就是无参数之构造函数）。这个条件常为人所忽略，但它是必要的，因为若一个对象来自文件，MFC 必须先动态生成它，而且在没有任何参数的情况下调用其构造函数，然后才从文件中读出对象资料。

如此，让我们再复习一次本例之CStroke，看看是否符合上述五大条件：

```
// in SCRIBBLEDOC.H
class CStroke : public CObject  // 衍生自CObject（條件1）
{
public:
       CStroke(UINT nPenWidth);

protected:
       CStroke();               // 擁有一個 default constructor （條件5）
       DECLARE_SERIAL(CStroke) // 使用 SERIAL 巨集 （條件2）

protected:
       UINT  m_nPenWidth;
public:
       CArray<CPoint,CPoint>  m_pointArray;

public:
       virtual void Serialize(CArchive& ar);  // 改寫 Serialize 函式（條件4）
};

// in SCRIBBLEDOC.CPP
IMPLEMENT_SERIAL(CStroke, CObject, 1)   // 使用 SERIAL 巨集（條件3）

CStroke::CStroke()   // 擁有一個 default constructor（條件5）
{
       // This empty constructor should be used by serialization only
}
```

```
void CStroke::Serialize(CArchive& ar)  // 改寫 Serialize 函式（條件4）
{
       CObject::Serialize(ar); // 手冊上告訴我們最好先呼叫此函式。
                               // 目前 MFC 版本中它是空函式，所以不呼叫也沒關係。
       if (ar.IsStoring())
       {
              ar << (WORD)m_nPenWidth;
              m_pointArray.Serialize(ar);
       }
       else
       {
              WORD w;
              ar >> w;
              m_nPenWidth = w;
              m_pointArray.Serialize(ar);
       }
}
```

CObject类

为什么绝大部分的 MFC 类，以及许多你自己的类，都要从 CObject 派生下来呢？因为当一个类派生自 CObject，它也就继承了许多重要的性质。CObject 这个「老祖宗」至少提供两个机能（两个虚函数）：IsKindOf 和 IsSerializable。

IsKindOf

当Framework 掌握「类型录网」这张王牌，要设计出 IsKindOf 根本不是问题。所谓IsKindOf 就是RTTI的化身，用白话说就是「xxx 对象是一种xxx 类吗？」例如「长臂猿是一种哺乳类吗？」「蓝鲸是一种鱼类吗？」凡支持 RTTI 的程序就必须接受这类询问，并对前者回答Yes，对后者回答No。

下面是 CObject::IsKindOf 虚函数的原始代码：

```
// in OBJCORE.CPP
BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
{
        // simple SI case
        CRuntimeClass* pClassThis = GetRuntimeClass();
        return pClassThis->IsDerivedFrom(pClass);
}

BOOL CRuntimeClass::IsDerivedFrom(const CRuntimeClass* pBaseClass) const
{
        // simple SI case
        const CRuntimeClass* pClassThis = this;
        while (pClassThis != NULL)
        {
                if (pClassThis == pBaseClass)
                        return TRUE;
                pClassThis = pClassThis->m_pBaseClass;
        }
        return FALSE;      // walked to the top, no match
}
```

这项作为，也就是在图8-9中借着m_pBaseClass 寻根。只要在寻根过程中比对成功，就传回TRUE，否则传回FALSE。 而你知道，图8-9 的「类型录网」是靠DECLARE_DYNAMIC和IMPLEMENT_DYNAMIC宏构造起来的。第 3 章的「RTTI」一节对此多有说明。

IsSerializable

一个类若要能够进行Serialization 动作，必须准备Serialize 函数，并且在「类型录网」中自己的那个CRuntimeClass元素里的schema字段里设立 0xFFFF 以外的号代码，代表数据格式的版本（这样才能提供机会让设计较佳的 Serialize 函数能够区分旧版数据或新版资料，避免牛头不对马嘴的困惑）。这些都是DECLARE_SERIAL和IMPLEMENT_SERIAL宏的责任范围。

CObject提供了一个虚函数，让程序在执行时期判断某类的schema号代码是否为0xFFFF，藉此得知它是否可以Serialize：

```
BOOL CObject::IsSerializable() const
{
    return (GetRuntimeClass()->m_wSchema != 0xffff);
}
```

CObject::Serialize

这是一个虚函数。每一个希望具备Serialization 能力的类都应该改写它。事实上Wizard 为我们做出来的程序代码中也都会自动加上这个函数的调用动作。MFC手册上总是说，每一个你所改写的Serialize函数都应该在第一时间调用此一函数，那么是不是CObject::Serialize 之中有什么重要的动作？

```
// in AFX.INL
_AFX_INLINE void CObject::Serialize(CArchive&)
{ /* CObject does not serialize anything by default */ }
```

不，什么也没有。所以，现阶段（至少截至 MFC 4.0）你可以不必理会手册上的谆谆告诲。然而，Microsoft 很有可能改变CObject::Serialize 的内容，届时没有遵循告诲的人恐怕就后悔了。

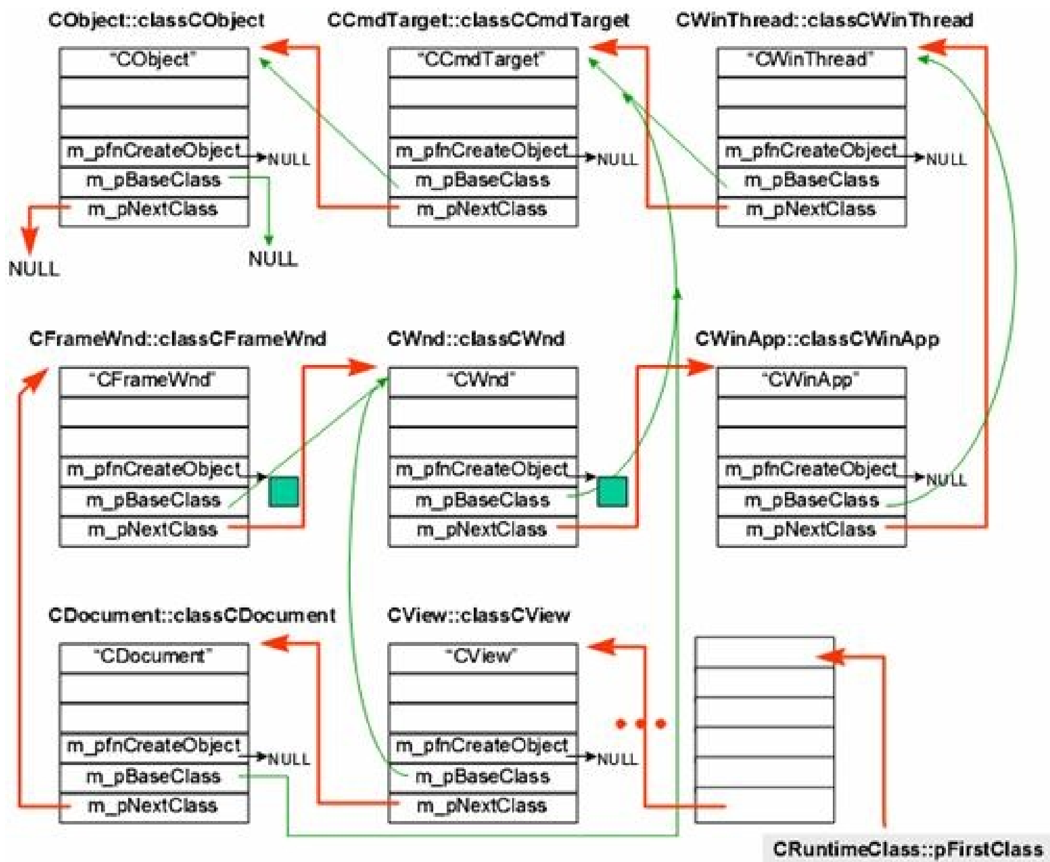


图8-9 DECLARE 和*IMPLEMENT*宏合力构造起这张网

于是RTTI和Dynamic Creation和Serialization等机能便可轻易达成

CArchive 类

谈到Serialize 就不能不谈CArchive，因为serialize 的对象（无论读或写）是一个CArchive 对象，这一点相信你已经从上面数节讨论中熟悉了。基本上你可以想象archive相当于文件，不过它其实是文件之前的一个内存缓冲区。所以我们才会在前面的「台面下的 Serialize 奥秘」中看到这样的动作：

```
BOOL CDocument::OnSaveDocument(LPCTSTR lpszPathName)
{
   CFile* pFile = NULL;
   pFile = GetFile(lpszPathName, CFile::modeCreate |
                   CFile::modeReadWrite | CFile::shareExclusive, &fe);

   // 令 file 和 archive 產生關聯
    CArchive saveArchive(pFile, CArchive::store |
                         CArchive::bNoFlushOnDelete);
    ...
    Serialize(saveArchive);   // 對著 archive 做 serialize 動作
    ...
    saveArchive.Close();
    ReleaseFile(pFile, FALSE);
 }

BOOL CDocument::OnOpenDocument(LPCTSTR lpszPathName)
{
    CFile* pFile = GetFile(lpszPathName,
                           CFile::modeRead|CFile::shareDenyWrite, &fe);

    // 令 file 和 archive 產生關聯
    CArchive loadArchive(pFile, CArchive::load |
                         CArchive::bNoFlushOnDelete);
```

```
    // 令 file 和 archive 產生關聯
    CArchive loadArchive(pFile, CArchive::load |
                         CArchive::bNoFlushOnDelete);
    ...
    Serialize(loadArchive);   // 對著 archive 做 serialize 動作
    ...
    loadArchive.Close();
    ReleaseFile(pFile, FALSE);
}
```

`operator<<` 和 `operator>>`

CArchive 针对许多C++ 数据类型、Windows 数据类型以及CObject 派生类，定义 `operator<<` 和 `operator>>` 重载运算符：

```
// in AFX.H
class CArchive
{
public:
// Flag values
        enum Mode { store = 0, load = 1, bNoFlushOnDelete = 2, bNoByteSwap = 4 };
        CArchive(CFile* pFile, UINT nMode, int nBufSize = 4096, void* lpBuf = NULL);
        ~CArchive();

// Attributes
        BOOL IsLoading() const;
        BOOL IsStoring() const;
        BOOL IsByteSwapping() const;
        BOOL IsBufferEmpty() const;

        CFile* GetFile() const;
        UINT GetObjectSchema(); // only valid when reading a CObject*
        void SetObjectSchema(UINT nSchema);

        // pointer to document being serialized -- must set to serialize
        //  COleClientItems in a document!
        CDocument* m_pDocument;

// Operations
        UINT Read(void* lpBuf, UINT nMax);
        void Write(const void* lpBuf, UINT nMax);
        void Flush();
        void Close();
        void Abort();   // close and shutdown without exceptions

        // reading and writing strings
        void WriteString(LPCTSTR lpsz);
        LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
        BOOL ReadString(CString& rString);

public:
        // Object I/O is pointer based to avoid added construction overhead.
        // Use the Serialize member function directly for embedded objects.
        friend CArchive& AFXAPI operator<<(CArchive& ar, const CObject* pOb);

        friend CArchive& AFXAPI operator>>(CArchive& ar, CObject*& pOb);
        friend CArchive& AFXAPI operator>>(CArchive& ar, const CObject*& pOb);
```

```
        // insertion operations
        CArchive& operator<<(BYTE by);
        CArchive& operator<<(WORD w);
        CArchive& operator<<(LONG l);
        CArchive& operator<<(DWORD dw);
        CArchive& operator<<(float f);
        CArchive& operator<<(double d);

        CArchive& operator<<(int i);
        CArchive& operator<<(short w);
        CArchive& operator<<(char ch);
        CArchive& operator<<(unsigned u);

        // extraction operations
        CArchive& operator>>(BYTE& by);
        CArchive& operator>>(WORD& w);
        CArchive& operator>>(DWORD& dw);
        CArchive& operator>>(LONG& l);
        CArchive& operator>>(float& f);
        CArchive& operator>>(double& d);

        CArchive& operator>>(int& i);
        CArchive& operator>>(short& w);
        CArchive& operator>>(char& ch);
        CArchive& operator>>(unsigned& u);

        // object read/write
        CObject* ReadObject(const CRuntimeClass* pClass);
        void WriteObject(const CObject* pOb);
        // advanced object mapping (used for forced references)
        void MapObject(const CObject* pOb);

        // advanced versioning support
        void WriteClass(const CRuntimeClass* pClassRef);
        CRuntimeClass* ReadClass(const CRuntimeClass* pClassRefRequested = NULL,
               UINT* pSchema = NULL, DWORD* pObTag = NULL);
        void SerializeClass(const CRuntimeClass* pClassRef);
        ...
protected:
        // array/map for CObject* and CRuntimeClass* load/store
        UINT m_nMapCount;
        union
        {
               CPtrArray* m_pLoadArray;
               CMapPtrToPtr* m_pStoreMap;
        };
        // map to keep track of mismatched schemas
        CMapPtrToPtr* m_pSchemaMap;
        ...
};
```

这些重载运算符均定义于AFX.INL 文件中。另有些函数可能你会觉得眼熟，没错，它们在稍早的「台面下的Serialize奥秘」中已经出现过了，它们是 ReadObject、WriteObject、ReadClass、WriteClass。

各种类型的 `operator>>` 和 `operator<<` 重载运算符，正是为什么你可以将各种类型的资料（甚至包括CObject*）读出或写入archive 的原因。一个「C++ 类」（而非一般资料类型）如果希望有Serialization 机制，它的第一要件就是直接或间接派生自Object，为的是希望自CObject 继承下列三个运算符：

```
// in AFX.INL
_AFX_INLINE CArchive& AFXAPI operator<<(CArchive& ar, const CObject* pOb)
        { ar.WriteObject(pOb); return ar; }
_AFX_INLINE CArchive& AFXAPI operator>>(CArchive& ar, CObject*& pOb)

        { pOb = ar.ReadObject(NULL); return ar; }
_AFX_INLINE CArchive& AFXAPI operator>>(CArchive& ar, const CObject*& pOb)
        { pOb = ar.ReadObject(NULL); return ar; }
```

其中CArchive::WriteObject先把类的CRuntimeClass资讯写出， 再调用类的Serialize 函数。CArchive::ReadObject的行为类似，先把类的 CRuntimeClass 信息读入，再调用类的Serialize 函数。Serialize是 CObject的虚函数，因此你必须确定你的类改写的Serialize 函数的回返值和参数类型都符合CObject 中的声明：传回值为void，唯一一个参数为 CArchive&。

注意：CString、CRect、CSize、CPoint 并不派生自CObject，但它们也可以直接使用针对CArchive 的 `<<` 和 `>>` 运算符，因为它们自己设计了一套：

```
// in AFX.H
class CString
{
    friend CArchive& AFXAPI operator<<(CArchive& ar, const CString& string);
    friend CArchive& AFXAPI operator>>(CArchive& ar, CString& string);
    ...
};

// in AFXWIN.H
// Serialization
CArchive& AFXAPI operator<<(CArchive& ar, SIZE size);
CArchive& AFXAPI operator<<(CArchive& ar, POINT point);
CArchive& AFXAPI operator<<(CArchive& ar, const RECT& rect);
CArchive& AFXAPI operator>>(CArchive& ar, SIZE& size);
CArchive& AFXAPI operator>>(CArchive& ar, POINT& point);
CArchive& AFXAPI operator>>(CArchive& ar, RECT& rect);
```

一个类如果希望有Serialization 机制，它的第二要件就是使用 SERIAL宏。这个宏包容DYNCREATE 宏，并且在类的声明之中加上：

```
friend CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb);
```

在类的实现文件中加上：

```
CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb) \
{ pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
return ar; } \
```

如果我的类名为CStroke，那么经由

```
class CStroke : public CObject
{
    ...
    DECLARE_SERIAL(CStroke)
}
```

和

```
IMPLEMENT_SERIAL(CStroke, CObject, 1)
```

我就获得了两组和 CArchive 读写动作的关键性程序代码：

```
class CStroke : CObject
{
    ...
    friend CArchive& AFXAPI operator>>(CArchive& ar, CStroke* &pOb);
}
CArchive& AFXAPI operator>>(CArchive& ar, CStroke* &pOb)
{ pOb = (CStroke*) ar.ReadObject(RUNTIME_CLASS(CStroke));
return ar; }
```

好，你看到了，为什么只改写 `operator>>`，而没有改写 `operator<<`？原因是 WriteObject 并不需要CRuntimeClass 信息，但ReadObject 需要，因为在读完文件后还要做动态生成的动作。

效率考虑

我想你一定在前面解剖文件文件倾印代码时就注意到了，当文件文件内含有许多对象数据时，凡对象隶属同一类者，只有第一个对象才连同类的 CRuntimeClass 信息一并写入，此后同类之对象仅以一个代码表示，例如图 8-6c 中时而出现的 8001 代码。为了效率的考虑，这是有必要的。想想看，如果一张 Scribble 图形有成千上万个线条，难不成要写入成千上万个 CStroke 信息不成？在哈滴（Hard Disk）极为便宜的今天，考虑的重点并不是文件的大小，而是文件大小背后所影响的读写时间，以及网络传输时间。别忘了，一切桌上的东西都将跃于网上。

CArchive维护类信息的作法是，当它做输出动作，对象名称以及参考值被维护在一个map之中；当它做读入动作，它把对象维护在一个array 之中。CArchive 中的成员变量 m_pSchemaMap 就是为此而来：

```
union
{
    CPtrArray* m_pLoadArray;
    CMapPtrToPtr* m_pStoreMap;
};
// map to keep track of mismatched schemas
CMapPtrToPtr* m_pSchemaMap;
```

自定 SERIAL 宏给抽象类使用

你是知道的，所谓抽象类就是包含纯虚函数的类，所谓纯虚函数就是只有声明没有定义的虚函数。所以，你不可能将抽象类具现化（instantiated）。那么，IMPLEMENT_SERIAL 展开所得的这段代码：

```
CObject* PASCAL class_name::CreateObject() \
{ return new class_name; } \
```

面对如果一个抽象类 class_name 就行不通了，编译时会产生错误消息。这时你得自行定义宏如下：

```
#define IMPLEMENT_SERIAL_MY(class_name, base_class_name, wSchema) \
_IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, NULL) \
CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb) \
{ pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
return ar; } \
```

也就是，令CreateObject函数为NULL，这才能够使用于抽象类之中。

在 CObList 中加入 CStroke 以外的类

Scribble Document倾印代码中的那个代表「旧类」的8001一直令我如坐针毡。不知道什么情况下会出现8002？或是8003？或是什么其它东东。因此，我打算做点测试。除了CStroke，我打算再加上 CRectangle 和 CCircle 两个类，并把其对象挂到CObList中。这个修改纯粹为了测试不同类写到文件文件中会造成什么后果，没有考虑使用者介面或任何外围因素，我并不是真打算为 Scribble 加上画四方形和画圆形的功能（不过如果你喜欢，这倒是能够给你作为一个导引），所以我把 Step1 拷贝一份，在拷贝版上做文章。

首先我必须声明 CCircle 和 CRectangle。在新文件中做这件事当然可以，但考虑到简化问题，以及它们与 CStroke 可能会有彼此前置参考的情况，我还是把它们放在原有的 ScribbleDoc.h中好了。为了能够"Serialize"，它们都必须派生自CObject，使用 DECLARE_SERIAL宏，并改写Serialize 虚函数，而且拥有default constructor。

CRectange有一个成员变量CRect m_rect，代表四方形的四个点；CCircle 有一个成员变量 CPoint m_center 和一个成员变量UINT m_radius，代表圆心和半径：

```
#0001  class CRectangle : public CObject
#0002  {
#0003  public:
#0004          CRectangle(CRect rect);
#0005
#0006  protected:
#0007          CRectangle();
#0008          DECLARE_SERIAL(CRectangle)
#0009
#0010  // Attributes
#0011          CRect m_rect;
#0012
```

```
#0013  public:
#0014          virtual void Serialize(CArchive& ar);
#0015  };
#0016
#0017  class CCircle : public CObject
#0018  {
#0019  public:
#0020          CCircle(CPoint center, UINT radius);
#0021
#0022  protected:
#0023          CCircle();
#0024          DECLARE_SERIAL(CCircle)
#0025
#0026  // Attributes
#0027          CPoint             m_center;
#0028          UINT               m_radius;
#0029
#0030  public:
#0031          virtual void Serialize(CArchive& ar);
#0032  };
```

接下来我必须在 ScribbleDoc.cpp 中使用 IMPLEMENT_SERIAL 宏，并定义成员函数。手册上要求每一个Serializable 类都应该准备一个空的构造函数（default constructor）。照着做吧，免得将来遗憾：

```
#0001  IMPLEMENT_SERIAL(CRectangle, CObject, 1)
#0002
#0003  CRectangle::CRectangle()
#0004  {
#0005     // this empty constructor should be used by serialization only
#0006  }
#0007
#0008  CRectangle::CRectangle(CRect rect)
#0009  {
#0010        m_rect = rect;
#0011  }
#0012
#0013  void CRectangle::Serialize(CArchive& ar)
#0014  {
#0015        if (ar.IsStoring())
#0016        {
#0017              ar << m_rect;
#0018        }
#0019        else
#0020        {
#0021              ar >> m_rect;
#0022        }
#0023  }
#0024
#0025  IMPLEMENT_SERIAL(CCircle, CObject, 1)
#0026
#0027  CCircle::CCircle()
#0028  {
#0029     // this empty constructor should be used by serialization only
#0030  }
#0031
#0032  CCircle::CCircle(CPoint center, UINT radius)
#0033  {
#0034        m_radius = radius;
#0035        m_center = center;
#0036  }
#0037
#0038  void CCircle::Serialize(CArchive& ar)
#0039  {
#0040        if (ar.IsStoring())
#0041        {
#0042              ar << m_center;
#0043              ar << m_radius;
#0044        }
#0045        else
#0046        {
#0047              ar >> m_center;
#0048              ar >> m_radius;
#0049        }
#0050  }
```

接下来我应该改变使用者接口，加上选单或工具列，以便在涂鸦过程中得随时加上一个四方形或一个圆圈。但我刚才说了，我只是打算做个小小的文件文件格式测试而已，所以简单化是我的最高指导原则。我打算搭现有之使用者接口的便车，也就是每次鼠标左键按下开始一条线条之后，再 new 一个四方形和一个圆形，并和线条一起加入 CObList 之中，然后才开始接受左键的坐标...。所以，我修改 CScribDoc::NewStroke 函数如下：

```
#0001  CStroke* CScribDoc::NewStroke()
#0002  {
#0003      CStroke* pStrokeItem = new CStroke(m_nPenWidth);
#0004      CRectangle* pRectItem = new CRectangle(CRect(0x11,0x22,0x33,0x44));
#0005      CCircle* pCircleItem = new CCircle(CPoint(0x55,0x66),0x77);
#0006      m_strokeList.AddTail(pStrokeItem);
#0007      m_strokeList.AddTail(pRectItem);
#0008      m_strokeList.AddTail(pCircleItem);
#0009
#0010      SetModifiedFlag();  // Mark the document as having been modified,
#0011                          // for purposes of confirming File Close.
#0012      return pStrokeItem;
#0013  }
```

并将scribbledoc.h 中的m_strokeList 修改为：

```
CTypedPtrList<CObList, CObject*> m_strokeList;
```

重新编译链接，获得结果如图 8-10a。图8-10b 对此结果有详细的剖析。

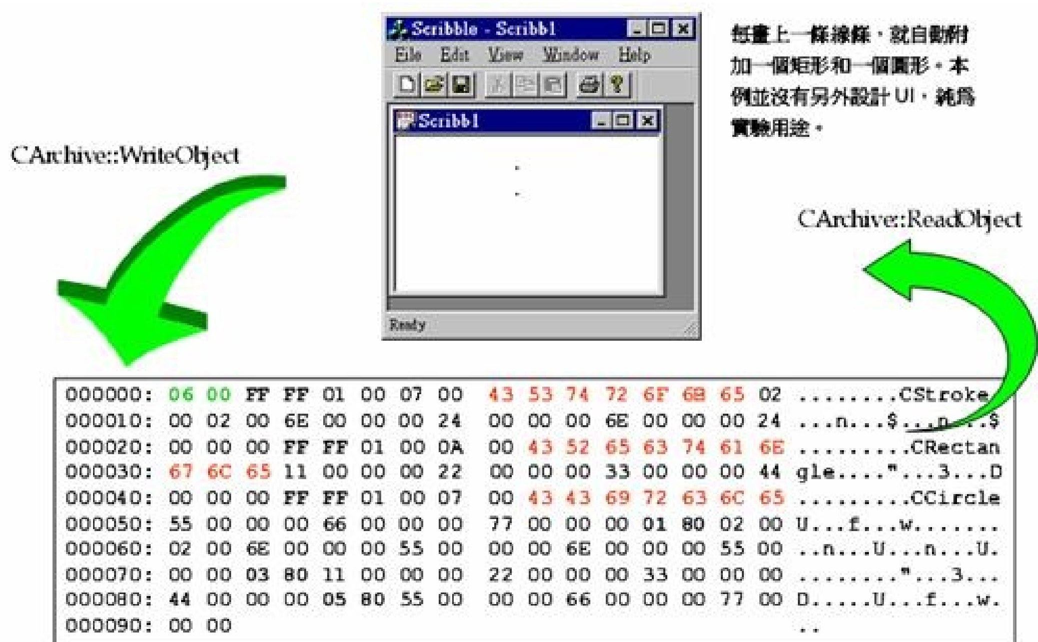
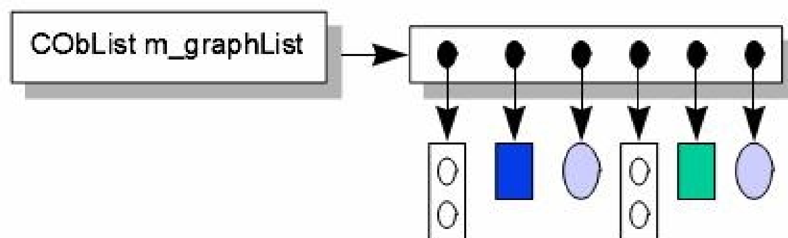


图8-10a TEST.SCB 文件内容，文件全长146个字节



每次鼠标左键按下，开始一条线条，图8-10a中的程序立刻new一个四方形和一个圆形，并和线条一起加入CObList 之中，然后才开始接受左键的坐标。所以图8-10a 的执行画面造成本图的数据结构。

數值（hex）	說明
0006	表示此檔有六個 *CObList* 元素
FFFF	FFFF 亦即 -1，表示 New Class Tag
0001	這是 Schema no.，代表資料的版本號碼
0007	表示後面接著的「類別名稱」有 7 個字元
43 53 74 72 6F 6B 65	"CStroke"（類別名稱）的 ASCII 碼
0002	第一條線條的寬度
0002	第一條線條的點陣列大小（點數）
00000066,0000001B	第一條線條的第一個點座標
00000066,0000001B	第一條線條的第二個點座標
FFFF	FFFF 亦即 -1，表示 New Class Tag
0001	這是 Schema no.，代表資料的版本號碼。
000A	後面接著的「類別名稱」有 Ah 個字元。
43 52 65 63 74 61 6E 67 6C 65	"CRectangle"（類別名稱）的 ASCII 碼。
00000011	第一個四方形的左
00000022	第一個四方形的上
00000033	第一個四方形的右
00000044	第一個四方形的下
FFFF	FFFF 亦即 -1，表示 New Class Tag
0001	這是 Schema no.，代表資料的版本號碼。
0007	後面接著的「類別名稱」有 7 個字元。
43 43 69 72 63 6C 65	"CCircle"（類別名稱）的 ASCII 碼。
00000055	第一個圓形的中心點 X 座標
00000066	第一個圓形的中心點 Y 座標
00000077	第一個圓形的半徑
8001	這是 (*wOldClassTag* | *nClassIndex*) 的組合結果，表示接下來的物件使用索引爲 1 的舊類別。
0002	第二條線條的寬度
0002	第二條線條的點陣列大小（點數）

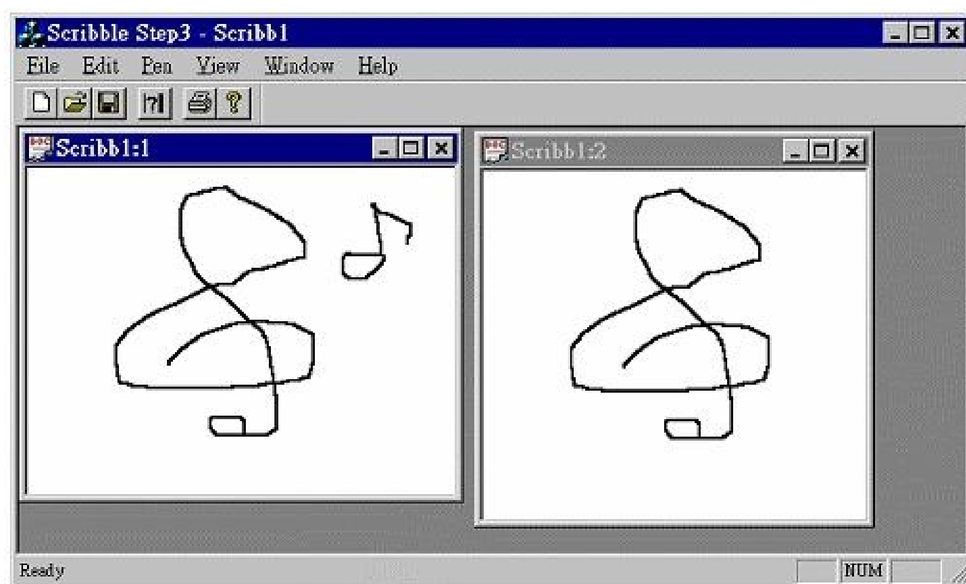
00000066,00000031	第二條線條的第一個點座標
00000066,00000031	第二條線條的第二個點座標
8003	這是 (*wOldClassTag* | *nClassIndex*) 的組合結果，表示接下來的物件使用索引爲 3 的舊類別。
00000011	第二個四方形的左
00000022	第二個四方形的上
00000033	第二個四方形的右
00000044	第二個四方形的下
8005	這是 (*wOldClassTag* | *nClassIndex*) 的組合結果，表示接下來的物件使用索引爲 5 的舊類別。
00000055	第二個圓形的中心點 X 座標
00000066	第二個圓形的中心點 Y 座標
00000077	第二個圓形的半徑

图8-10b TEST.SCB文件内容剖析。别忘了Intel采用"little-endian"

位组排列方式，每一字组的前后字节系颠倒放置。本图已将之摆正。

Document 与 View 交流——为 Step4 做准备

虽然 Scribble Step1 已经可以正常工作，有些地方仍值得改进。在一个子窗口上作画，然后选按【Window/New Window】，会蹦出一个新的子窗口，内有第一个子窗口的图形，同时，第一个子窗口的标题加上 :1 字样，第二个子窗口的标题则有 :2 字样。这是Document/View架构带给我们的礼物，换句话说，想以多个窗口观察同一份数据，程序员不必负担什么任务。但是，如果此后使用者在其中一个子窗口上作画而不缩放窗口尺寸的话（也就是没有产生 WM_PAINT），另一个子窗口内看不到新的绘图内容：



这不是好现象！一体的两面怎么可以不一致呢?!

那么，让「作用中的 View 窗口」以消息通知隶属同一份 Document 的其它「兄弟窗口」，是不是就可以解决这个问题？是的，而且 Framework 已经把这样的机制埋伏下去了。

CView 之中的三个虚函数：

- CView::OnInitialUpdate——负责View的初始化。
- CView::OnUpdate——当Framework 调用此函数，表示Document的内容已有变化。
- CView::OnDraw——Framework将在WM_PAINT 发生后，调用此函数。此函数应负责更新 View 窗口的内容。

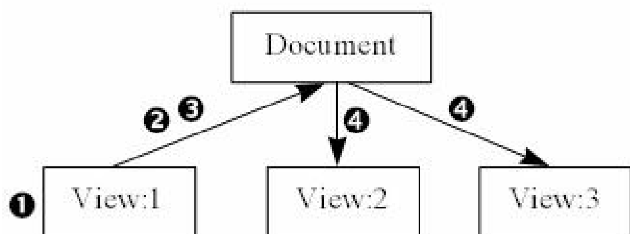
  这些函数往往成为程序员改写的目标。Scribble第一版就是因为只改写了其中的OnDraw函数，所以才有「多个 View 窗口不能同步更新」的缺失。想要改善这项缺失，我们必须改写 OnUpdate。

  让所有的 View 窗口「同步」更新数据的关键在于两个函数：

- CDocument::UpdateAllViews——如果这个函数执行起来，它会巡访所有隶属同一Document 的各个Views，找到一个就通知一个，而所谓「通知」就是调用View的OnUpdate 函数。
- CView::OnUpdate——这是一个虚函数，我们可以改写它，在其中设计绘图动作，也许全部重绘（这比较笨一点），也许想办法只绘必要的一小部分（这样速度比较快，但设计上比较复杂些）。

因此，当一个Document 的数据改变时，我们应该设法调用其 UpdateAllViews，通知所有的Views。什么时候Scribble 的数据会改变？答案是鼠标左键按下时！所以你可能猜测到，我打算在CView::OnLButtonDown 内调用CDocument::UpdateAllViews。这个猜测的立论点是对的而结果是错的，Scribble Step4 的作法是在 CView::OnButtonUp 内部调用它。

CView::OnUpdate 被调用，代表着 View 被告知：「嘿，Document 的内容已经改变了，请你准备修改你的显示画面」。如果你想节省力气，利用 Invalidate(TRUE) 把窗口整个设为重绘区（无效区）并产生 WM_PAINT，再让 CView::OnDraw 去伤脑筋算了。但是全部重绘的效率低落，程序看起来很笨拙，Step4 将有比较精致的作法。



- 1 使用者在 View:1 做动作（View 扮演使用者接口的第一线）。
- 2 View:1 调用 GetDocument，取得 Document 指针，更改数据内容。

- 3 View:1 调用 Document 的 UpdateAllViews。
- 4 View:2 和 View:3 的 OnUpdate 一一被调用起来，这是更新画面的时机。

图8-11 假设一份Document链接了三个Views

注意：在MFC手册或其它书籍中，你可能会看到像「View1 以消息通知 Document」或「Document 以消息通知 View2、View3」的说法。这里所谓的「消息」是面向对象学术界的术语，不要和 Windows 的消息混淆了。事实上整个过程中并没有任何一个Windows消息参与其中。

第 9 章　消息映射与命令绕行

> Message Mapping and Command Routing
>
> 消息映射机制与命令绕行，活像是米诺托斯的迷宫，是 MFC 最曲折幽深的神秘地带。

你已经从前一章中彻底了解了MFC程序极端重要的Document/View 架构。本章的重点有两个，第一个是修改程序的人机接口，增添选单项目和工具列按钮。这一部分藉Visual C++ 工具之助，非常简单，但是我们往往不知道该在程序的什么地方（哪一个类之中）处理来自选单和工具列的消息（也就是 WM_COMMAND 消息）。本章第二个重点就是要解决这个迷惑，我将对所谓的消息映射（Message Map）和命令绕行（Command Routing）机制做深入的讨论。这两个机制宛如 MFC 最曲折幽深的神秘地带，是把杂乱无章的Windows API 函数和 Windows 消息面向对象化的大功臣。

到底要解决什么

Windows 程序的本质系借着消息来维持脉动。每一个消息都有一个代码，并以WM_ 开头的常数表示之。消息在传统SDK程序方法中的流动以及处置方式，在第 1 章已经交待得很清楚。

各种消息之中，来自选单或工具列者，都以WM_COMMAND 表示，所以这一类消息我们又称之为命令消息（Command Message），其wParam 记录着此一消息来自哪一个选单项目。

除了命令消息，还有一种消息也比较特殊，出现在对话框函数中，是控制组件（controls）传送给父窗口（即对话框）的消息。虽然它们也是以WM_COMMAND 为外衣，但特别归类为「notification 消息」。

注意：Windows 95 新的控制组件（所谓的common controls）不再传送 WM_COMMAND消息给对话框，而是送 WM_NOTIFY。这样就不会纠缠不清了。但为了回溯相容，旧有的控制组件（如 edit、list box、combo box...）都还是传送WM_COMMAND。

消息会循着Application Framework 规定的路线，游走于各个对象之间，直到找到它的依归（消息处理函数）。找不到的话，Framework 最终就把它交给::DefWindowProc 函数去处理。

但愿你记忆犹新，第 6 章曾经挖掘 MFC 原始代码，得知 MFC 在为我们产生窗口之前，如果我所指定的窗口类是 NULL，MFC 会自动先注册一个适当的窗口类。这个类在动态链接、除错版、非 Unicode 环境的情况下，可能是下列五种窗口类之一：

- "AfxWnd42d"
- "AfxControlBar42d"

- "AfxMDIFrame42d"
- "AfxFrameOrView42d"
- "AfxOleControl42d"

每一个窗口类有它自己的窗口函数。根据 SDK 的基础，我们推想，不同窗口所获得的消息，应该由不同的窗口函数来处理。如果都没有能够处理，最后再交由Windows API函数::DefWindowProc 处理。

这是很直觉的想法，而且对于一般消息（如 WM_MOVE、WM_SIZE、WM_CREATE 等）也是天经地义的。但是今天 Application Framework 比传统的SDK 程序多出了一个Document/View 架构，试想，如果选单上有个命令项关乎文件的处理，那么让这个命令消息流到Document 类去不是最理想吗？一旦流入 Document 大本营，我们（程序员）就可以很方便地取得Document 成员变量、调用Document成员函数，做爱做的事。

但是Document 不是窗口，也没有对应的窗口类，怎么让消息能够七拐八弯地流往Document 类去？甚至更往上流向 Application 类去？这就是所谓的命令绕行机制！

而为了让消息的流动有线路可循，MFC 必须做出一个巨大的网，实现所有可能的路线，这个网就是所谓的消息映射地图（Message map）。最后，MFC 还得实现一个消息推动引擎，让消息依 Framework 的意旨前进，该拐的时候拐，该弯的时候弯，这个捕获机制埋藏在各个类的WindowProc、OnCommand、OnCmdMsg、DefWindowProc 虚函数中。

没有命令绕行机制，Document/View 架构就像断了条胳臂，会少掉许多功用。

很快你就会看到所有的秘密。很快地，它们统统不再对你构成神秘。

消息分类

Windows 的消息都是以WM*xxx* 为名，*WM* 的意思是"Windows Message"。消息可以是来自硬件的「输入消息」，例如 WM_LBUTTONDOWN，也可以是来自 USER 模块的「窗口管理消息」，例如 WM_CREATE。这些消息在 MFC 程序中都是隐晦的（我的意思是不像在 SDK 程序中那般显明），我们不必在 MFC 程序中撰写 switch case 指令，不必一一识别并处理由系统送过来的消息；所有消息都将依循 Framework 制定的路线，并参照路中是否有拦路虎（你的消息映射表格）而流动。WM_PAINT 一定流往你的OnPaint函数去，WM_SIZE 一定流往你的 OnSize 函数去。

所有的消息在 MFC 程序中都是暗潮汹涌，但是表面无波。

MFC 把消息分为三大类：

- 命令消息（WM_COMMAND）：命令消息意味着「使用者命令程序做某些动作」。

凡由UI对象产生的消息都是这种命令消息，可能来自选单或加速键或工具列按钮，并且都以WM_COMMAND 呈现。如何分辨来自各处的命令消息？SDK程序主要靠消息的wParam 辨识之，MFC程序则主要靠选单项目的识别代码（menu ID）辨识之——两者其实是相同的。

什么样的类有资格接受命令消息？凡派生自 CCmdTarget 之类，皆有资格。从command target的字面意义可知，这是命令消息的目的地。也就是说，凡派生自CCmdTarget者，它的骨子里就有了一种特殊的机制。把整张MFC类阶层图摊开来看，几乎构造应用程序的最重要的几个类都派生自 CCmdTarget，剩下的不能接收消息的，是像CFile、CArchive、CPoint、CDao（数据库）、Collection Classes（纯粹数据处理）、GDI 等等「非主流」类。

- 标准消息——除WM*COMMAND* 之外，任何以*WM* 开头的都算是这一类。任何派生自CWnd 之类，均可接收此消息。
- Control Notification - 这种消息由控制组件产生，为的是向其父窗口（通常是对话框）通知某种情况。例如当你在ListBox 上选择其中一个项目，ListBox 就会产生LBN_SELCHANGE传送给父窗口。这类消息也是以WM_COMMAND 形式呈现。

万流归宗 Command Target（CCmdTarget）

你可以在程序的许多类之中设计拦路虎（我是指「消息映射表格」），接收并处理消息。只要是CWnd 派生类，就可以拦下任何Windows 消息。与窗口无关的MFC类（例如CDocument和CWinApp）如果也想处理消息，必须派生自 CCmdTarget，并且只可能收到WM_COMMAND 命令消息。

会产生命令消息的，不外就是 UI 对象：选单项目和工具列按钮都是。命令消息必须有一个对应的处理函数，把消息和其处理函数「绑」在一块儿，这动作称为Command Binding，这个动作将由一堆宏完成。通常我们不直接手工完成这些宏内容，也就是说我们并不以文字编辑器一行一行地撰写相关的代码，而是藉助于 ClassWizard。

一个 Command Target 对象如何知道它可以处理某个消息？答案是它会看看自己的消息映射表。消息映射表使得消息和函数的对映关系形成一份表格，进而全体形成一张网，当Command Target 对象收到某个消息，便可由表格得知其处理函数的名称。

三个奇怪的宏，一张巨大的网

早在本书第 1 章我就介绍过消息映射的雏形了，不过那是小把戏，不登大雅之堂。第 3 章以DOS 程序仿真消息映射，就颇有可观之处，因为那是「偷」MFC 的原始代码完成的，可以说具体而微。

试着思考这个问题：C++ 的继承与多态性质，使派生类与基类的成员函数之间有着特殊的关联。但这当中并没有牵扯到 Windows 消息。的确，C++ 语言完全没有考虑Windows 消息这一回事（那当然）。如何让 Windows 消息也能够在面向对象以及继承性质中扮演一个角色？既然语言没有支持，只好自求多福了。消息映射机制的三个相关宏就是 MFC 自求多福的结果。

「消息映射」是 MFC 内建的一个消息分派机制，只要利用数个宏以及固定形式的写法，类似填表格，就可以让 Framework 知道，一旦消息发生，该循哪一条路递送。每一个类只能拥有一个消息映射表格，但也可以没有。下面是 Scribble Document 建立消息映射表的动作：

首先你必须在类声明文件（.H）声明拥有消息映射表格：

```
class CScribbleDoc : public CDocument
{
    ...
    DECLARE_MESSAGE_MAP()
};
```

然后在类实现文件（.CPP）实现此一表格：

```
BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)

//{{AFX_MSG_MAP(CScribbleDoc)
ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
ON_COMMAND(ID_PEN_THICK_OR_THIN, OnPenThickOrThin)
...
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

这其中出现三个宏。第一个宏 BEGIN*MESSAGE_MAP* 有两个参数，分别是拥有此消息映射表之类，及其父类。第二个宏是 *ON_COMMAND*，指定命令消息的处理函数名称。第三个宏 *END_MESSAGE_MAP* 作为结尾记号。至于夹在*BEGIN* 和END_ 之中奇奇怪怪的说明符号//}} 和 //{{，是ClassWizard 产生的，也是用来给它自己看的。记住，前面我就说了，很少人会自己亲手键入每一行代码，因为 ClassWizard 的表现相当不俗。

夹在BEGIN 和*END* 之中的宏，除了ON_COMMAND，还可以有许多种。标准的Windows 消息并不需要由我们指定处理函数的名称。标准消息的处理函数，其名称也是「标准」的（预设的），像是：

宏名称	对映消息	消息处理函数

ON_WM_CHAR	WM_CHAR	OnChar
ON_WM_CLOSE	WM_CLOSE	OnClose
ON_WM_CREATE	WM_CREATE	OnCreate
ON_WM_DESTROY	WM_DESTROY	OnDestroy
ON_WM_LBUTTONDOWN	WM_LBUTTONDOWN	OnLButtonDown
ON_WM_LBUTTONUP	WM_LBUTTONUP	OnLButtonUp
ON_WM_MOUSEMOVE	WM_MOUSEMOVE	OnMouseMove
ON_WM_PAINT	WM_PAINT	OnPaint

...

DECLARE_MESSAGE_MAP 宏

消息映射的本质其实是一个巨大的数据结构，用来为诸如 WM_PAINT 这样的标准消息决定流动路线，使它得以流到父类去；也用来为 WM_COMMAND 这个特殊消息决定流动路线，使它能够七拐八弯地流到类阶层结构的旁支去。

观察机密的最好方法就是挖掘原始代码：

```
// in AFXWIN.H
#define DECLARE_MESSAGE_MAP() \
private: \
        static const AFX_MSGMAP_ENTRY _messageEntries[]; \
protected: \
        static AFX_DATA const AFX_MSGMAP messageMap; \
        virtual const AFX_MSGMAP* GetMessageMap() const; \
```

注意：static 修饰词限制了数据的配置，使得每个「类」仅有一份数据，而不是每一个「对象」各有一份数据。

我们看到两个陌生的类型：AFX_MSGMAP_ENTRY和AFX_MSGMAP。继续挖原始代码，发现前者是一个struct：

```
// in AFXWIN.H
struct AFX_MSGMAP_ENTRY
{
        UINT nMessage;   // windows message
        UINT nCode;      // control code or WM_NOTIFY code
        UINT nID;        // control ID (or 0 for windows messages)
        UINT nLastID;    // used for entries specifying a range of control id's
        UINT nSig;       // signature type (action) or pointer to message #
        AFX_PMSG pfn;    // routine to call (or special value)
};
```

很明显你可以看出它的最主要作用，就是让消息nMessage对应于函数 pfn。其中pfn 的数据类型 AFX_PMSG 被定义为一个函数指针：

```
typedef void (AFX_MSG_CALL CCmdTarget::*AFX_PMSG)(void);
```

出现在 DECLARE_MESSAGE_MAP宏中的另一个struct，AFX_MSGMAP，定义如下：

```
// in AFXWIN.H
struct AFX_MSGMAP
{
        const AFX_MSGMAP* pBaseMap;
        const AFX_MSGMAP_ENTRY* lpEntries;
};
```

其中pBaseMap是一个指向「基类之消息映射表」的指针，它提供了一个走访整个继承串链的方法，有效地实现出消息映射的继承性。派生类将自动地「继承」其基类中所处理的消息，意思是，如果基类处理过 A 消息，其派生类即使未设计 A 消息之消息映射表项目，也具有对 A 消息的处理能力。当然啦，派生类也可以针对 A 消息设计自己的消息映射表项。

喝，真像虚函数！但Message Map没有虚函数所带来的巨大的overhead（额外负担）透过 DECLARE_MESSAGE_MAP 这么简单的一个宏，相当于为类声明了图9-1的数据类型。注意，只是声明而已，还没有真正的实体。

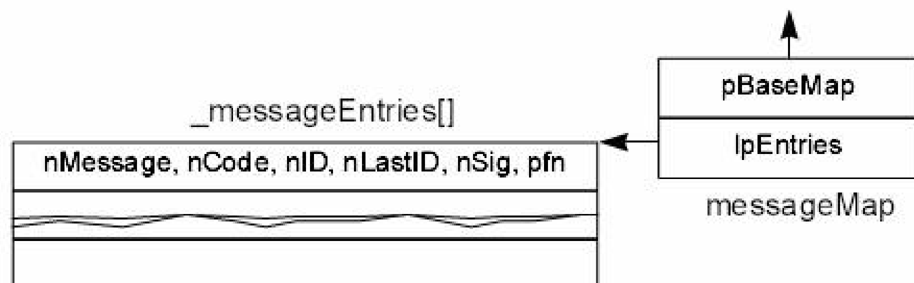


图9-1 DECLARE_MESSAGE_MAP宏相当于声明了这样的数据结构

消息映射网的形成：BEGIN ON END宏

前置准备工作完成了，接下来的课题是如何实现并填充图 9-1 的数据结构内容。当然你马上就猜到了，使用的是另一组宏：

```
BEGIN_MESSAGE_MAP(CMyView, CView)
ON_WM_PAINT()
ON_WM_CREATE()
...
END_MESSAGE_MAP()
```

奥秘还是在原始代码中：

```
// 以下原始代码在 AFXWIN.H
```

```
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
        const AFX_MSGMAP* theClass::GetMessageMap() const \
                { return &theClass::messageMap; } \
        AFX_DATADEF const AFX_MSGMAP theClass::messageMap = \
        { &baseClass::messageMap, &theClass::_messageEntries[0] }; \
        const AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
        { \

#define END_MESSAGE_MAP() \
                {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
        }; \
```

注意：AfxSig*end* 在*AFXMSG*.H中被定义为0

```
// 以下原始代码在 AFXMSG_.H
```

```
#define ON_COMMAND(id, memberFxn) \
        { WM_COMMAND, CN_COMMAND, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },

#define ON_WM_CREATE() \
        { WM_CREATE, 0, 0, 0, AfxSig_is, \
                (AFX_PMSG)(AFX_PMSGW)(int (AFX_MSG_CALL CWnd::*)(LPCREATESTRUCT))OnCreate },
#define ON_WM_DESTROY() \
        { WM_DESTROY, 0, 0, 0, AfxSig_vv, \
                (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(void))OnDestroy },
#define ON_WM_MOVE() \
        { WM_MOVE, 0, 0, 0, AfxSig_vvii, \
                (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(int, int))OnMove },
#define ON_WM_SIZE() \
        { WM_SIZE, 0, 0, 0, AfxSig_vwii, \
                (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, int, int))OnSize },
#define ON_WM_ACTIVATE() \
        { WM_ACTIVATE, 0, 0, 0, AfxSig_vwWb, \
                (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, CWnd*,
BOOL))OnActivate },
#define ON_WM_SETFOCUS() \
        { WM_SETFOCUS, 0, 0, 0, AfxSig_vW, \
                (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(CWnd*))OnSetFocus },
#define ON_WM_PAINT() \
        { WM_PAINT, 0, 0, 0, AfxSig_vv, \
                (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(void))OnPaint },
#define ON_WM_CLOSE() \
        { WM_CLOSE, 0, 0, 0, AfxSig_vv, \
                (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(void))OnClose },
...
```

于是，这样的宏：

```
BEGIN_MESSAGE_MAP(CMyView, CView)
ON_WM_CREATE()
ON_WM_PAINT()
END_MESSAGE_MAP()
```

便被展开成为这样的代码：

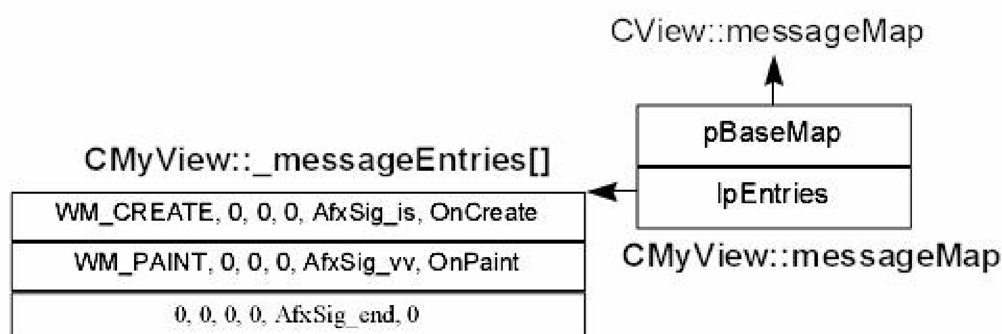
```
const AFX_MSGMAP* CMyView::GetMessageMap() const
        { return &CMyView::messageMap; }
AFX_DATADEF const AFX_MSGMAP CMyView::messageMap =
{ &CView::messageMap, &CMyView::_messageEntries[0] };
const AFX_MSGMAP_ENTRY CMyView::_messageEntries[] =
{
    { WM_CREATE, 0, 0, 0, AfxSig_is, \
        (AFX_PMSG)(AFX_PMSGW)(int (AFX_MSG_CALL CWnd::*)(LPCREATESTRUCT))OnCreate },
    { WM_PAINT, 0, 0, 0, AfxSig_vv, \
        (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(void))OnPaint },
    {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 }
};
```

其中AFX_DATADEF和AFX_MSG_CALL又是两个看起来很奇怪的常数。你可以在两个文件中找到它们的定义：

```
// in \DEVSTUDIO\VC\MFC\INCLUDE\AFXVER_.H
#define AFX_DATA
#define AFX_DATADEF
// in \DEVSTUDIO\VC\MFC\INCLUDE\AFXWIN.H
#define AFX_MSG_CALL
```

显然它们就像afx_msg一样（我曾经在第 6 章的HellpMFC 原始代码一出现之后解释过），都只是个"intentional placeholder"（刻意保留的空间），可能在将来会用到，目前则为「无物」。

以图表示BEGIN ON END宏的结果为：



注意：图中的 AfxSig*vv* 和*AfxSig_is* 都代表签名符号（*Signature*）。这些常数在 *AFXMSG*.H 中定义，稍后再述。

前面我说过了，所有能够接收消息的类，都应该派生自 CCmdTarget。那么我们这么推论应该是合情合理的：每一个派生自CCmdTarget的类都应该有DECLARE/*BEGIN*/END_ 宏组？

唔，错了，CWinThread 就没有！

可是这么一来，CWinApp 通往 CCmdTarget 的路径不就断掉了吗？呵呵，难道 CWinApp不能跳过 CWinThread 直接连上 CCmdTarget 吗？看看下面的 MFC 原始代码：

```
// in AFXWIN.H
class CWinApp : public CWinThread
{
...
        DECLARE_MESSAGE_MAP()
};

// in APPCORE.CPP
BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget) // 注意第二個參數是 CCmdTarget，
     //{{AFX_MSG_MAP(CWinApp)          // 而不是 CWinThread。
     // Global File commands

     ON_COMMAND(ID_APP_EXIT, OnAppExit)
     // MRU - most recently used file menu
     ON_UPDATE_COMMAND_UI(ID_FILE_MRU_FILE1, OnUpdateRecentFileMenu)
     ON_COMMAND_EX_RANGE(ID_FILE_MRU_FILE1, ID_FILE_MRU_FILE16, OnOpenRecentFile)
     //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

让我们看看具体的情况。图9-2 就是MFC的消息映射表。当你的派生类使用了DECLARE/*BEGIN*/END_宏，你也就把自己的消息映射表挂上去了——当然是挂在尾端。

如果没有把BEGIN_MESSAGE_MAP 宏中的两个参数（也就是类本身及其父类的名称）按照规矩来写，可能会发生什么结果呢？消息可能在不应该流向某个类时流了过去，在应该被处理时却又跳离了。总之，完美的机制有了破绽。程序没当掉算你幸运！

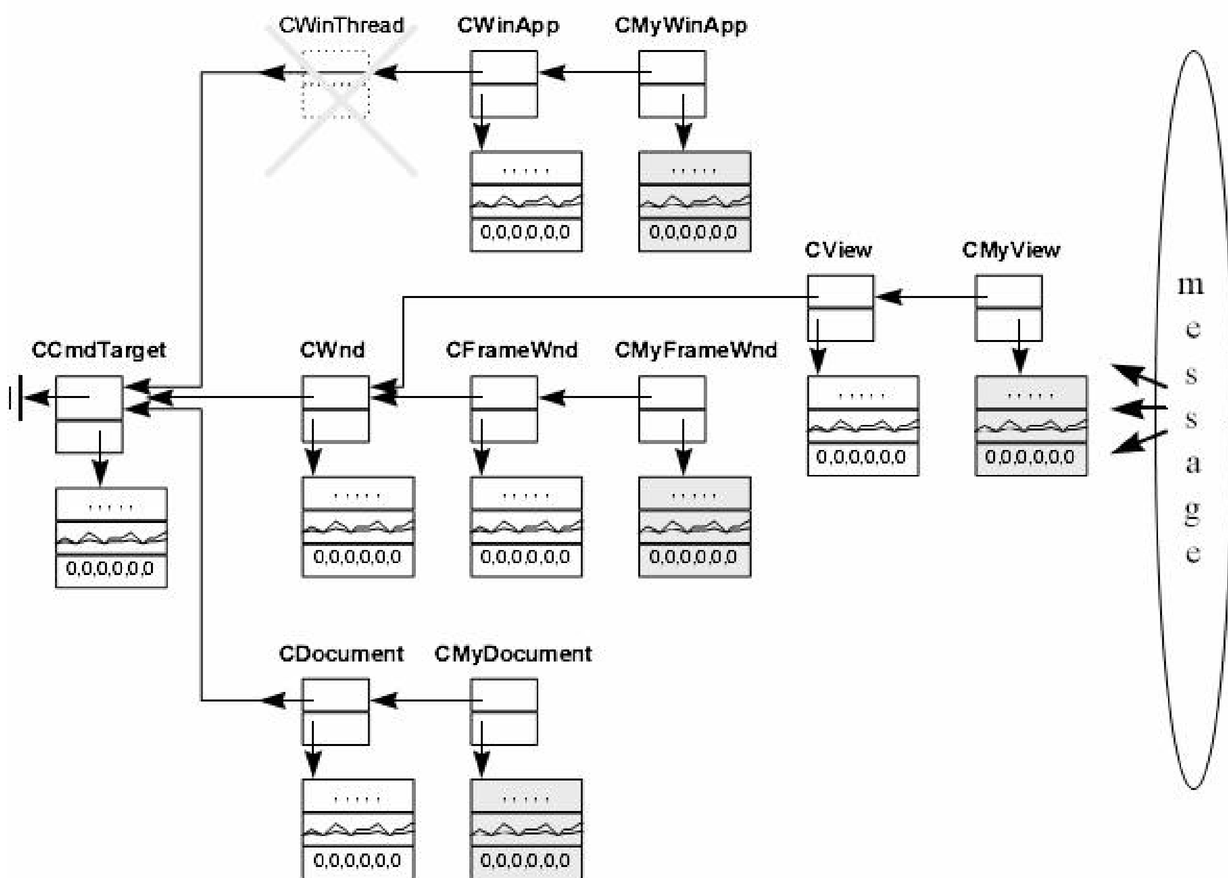


图9-2 MFC 消息映射表（也就是消息流动网）

我们终于了解，Message Map既可说是一套宏，也可以说是宏展开后所代表的一套数据结构；甚至也可以说 Message Map 是一种动作，这个动作，就是在刚刚所提的数据结构中寻找与消息相吻合的项目，从而获得消息的处理例程的函数指针。

虽然，C++ 程序员看到多态（Polymorphism），直觉的反应就是虚函数，但请注意，各个 Message Map 中的各个同名函数虽有多态的味道，却不是虚函数。乍想之下使用虚函数是合理的：你产生一个与窗口有关的C++ 类，然后为此窗口所可能接收的任何消息都提供一个对应的虚函数。这的确散发着 C++的味道和面向对象的精神，但现实与理想之间总是有些距离。

要知道，虚函数必须经由一个虚函数表（virtual function table，vtable）实现出来，每一个子类必须有它自己的虚函数表，其内至少有父类之虚函数表的内容复本（请参考第 2 章「类与对象大解剖」一节）。好哇，虚函数表中的每一个项目都是一个函数指针，价值 4 字节，如果基类的虚函数表有 100 个项目，经过 10 层继承，开枝散叶，总共需耗费多少内存在其中？最终，系统会被巨大的额外负担（overhead）拖垮！

这就是为什么 MFC 采用独特的消息映射机制而不采用虚函数的原因。

米诺托斯（**Minotauros**）与西修斯（**Theseus**）

截至目前我还有一些细节没有交待清楚，像是消息的比对动作、消息处理例程的调用动作、以及参数的传递等等，但至少现在可以先继续进行下去，我的目标瞄准消息唧筒（叫捕获也可以啦）。

窗口接收消息后，是谁把消息唧进消息映射网中？是谁决定消息该直直往父映射表走去？还是拐向另一条路（请回头看看图 9-2）？消息的绕行路线，以及MFC 的消息唧筒的设计，活像是米诺托斯的迷宫。不过别担心，我将扮演西修斯，让你免遭毒手。

米诺托斯（Minotauros），希腊神话里牛头人身的怪兽，为克里特岛国王迈诺斯之妻所生。迈诺斯造迷宫将米诺托斯藏于其中，每有人误入迷宫即遭吞噬。怪兽后为雅典王子西修斯（Theseus）所杀。

MFC2.5（注意，是2.5而非4.x）曾经在WinMain的第一个重要动作 AfxWinInit之中，自动为程序注册四个Windows窗口类，并且把窗口函数一致设为AfxWndProc：

```
//in APPINIT.CPP (MFC 2.5)
BOOL AFXAPI AfxWinInit(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR lpCmdLine, int nCmdShow)
{
    ...
    // register basic WndClasses  (以下開始註冊視窗類別)
    WNDCLASS wndcls;
    wndcls.lpfnWndProc = AfxWndProc;

    // Child windows - no brush, no icon, safest default class styles
    ...
    wndcls.lpszClassName = _afxWnd;
❶   if (!::RegisterClass(&wndcls))
          return FALSE;

    // Control bar windows
    ...
    wndcls.lpszClassName = _afxWndControlBar;
❷   if (!::RegisterClass(&wndcls))
          return FALSE;

    // MDI Frame window (also used for splitter window)
    ...
❸  if (!RegisterWithIcon(&wndcls, _afxWndMDIFrame, AFX_IDI_STD_MDIFRAME))
          return FALSE;

    // SDI Frame or MDI Child windows or views - normal colors
    ...
❹  if (!RegisterWithIcon(&wndcls, _afxWndFrameOrView, AFX_IDI_STD_FRAME))
          return FALSE;
    ...
}
```

下面是AfxWndProc 的内容：

```
// in WINCORE.CPP (MFC 2.5)
LRESULT CALLBACK AFX_EXPORT
AfxWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  CWnd* pWnd;

  pWnd = CWnd::FromHandlePermanent(hWnd);
  ASSERT(pWnd != NULL);
  ASSERT(pWnd->m_hWnd == hWnd);

  LRESULT lResult = _AfxCallWndProc(pWnd, hWnd, message, wParam, lParam);
  return lResult;
}

// Official way to send message to a CWnd
LRESULT PASCAL _AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT message,
                               WPARAM wParam, LPARAM lParam)
{
  LRESULT lResult;
  ...
  TRY
  {
     ...
     lResult = pWnd->WindowProc(message, wParam, lParam);
  }
  ...
  return lResult;
}
```

MFC 2.5的CWinApp::Run调用PumpMessage，后者又调用::DispatchMessage，把消息源源推往AfxWndProc（如上），最后流向 pWnd->WindowProc 去。拿 SDK 程序的本质来做比对，这样的逻辑十分容易明白。

MFC 4.x仍旧使用AfxWndProc作为消息唧筒的起点，但其间却隐藏了许多关节。

但愿你记忆犹新，第 6 章曾经说过，MFC 4.x 适时地为我们注册 Windows 窗口类（在第一次产生该种型式之窗口之前）。这些个 Windows 窗口类的窗口函数各是「窗口所对应之 C++ 类中的 DefWindowProc 成员函数」，请参考第 6 章「CFrameWnd::Create产生主窗口」一节。这就和 MFC 2.5 的作法（所有窗口类共享同一个窗口函数）有了明显的差异。那么，推动消息的心脏，也就是 CWinThread::PumpMessage 中调用的::DispatchMessage（请参考第 6 章「CWinApp::Run 程序生命的活水源头」一节），照说应该把消息唧到对应之C++ 类的 DefWindowProc 成员函数去。但是，我们发现MFC 4.x中仍然保有和MFC 2.5 相同的 AfxWndProc，仍然保有AfxCallWndProc，而且它们扮演的角色也没有变。

事实上，MFC 4.x 利用 hook，把看似无关的动作全牵联起来了。所谓 hook，是 Windows程序设计中的一种高阶技术。通常消息都是停留在消息队列中等待被所隶属之窗口抓取，如果你设立 hook，就可以更早一步抓取消息，并且可以抓取不属于你的消息，送往你设定的一个所谓「滤网函数（filter）」。

请查阅Win32 API 手册中有关于SetWindowsHook和SetWindowsHookEx两函数，以获得更多的hook 信息。（可参考Windows 95：A Developer’s Guide 一书第 6 章Hooks）

MFC 4.x的hook 动作是在每一个CWnd 派生类之对象产生之际发生，步骤如下：

```
// in WINCORE.CPP (MFC 4.x)
// 請回顧第6章「CFrameWnd::Create 產生主視窗」一節
BOOL CWnd::CreateEx(...)
{
    ...
    PreCreateWindow(cs);   //第6章曾經詳細討論過此一函式。
    AfxHookWindowCreate(this);
    HWND hWnd = ::CreateWindowEx(...);
    ...
}

// in WINCORE.CPP (MFC 4.x)
void AFXAPI AfxHookWindowCreate(CWnd* pWnd)
{
   ...
   pThreadState->m_hHookOldCbtFilter = ::SetWindowsHookEx(WH_CBT,
           _AfxCbtFilterHook, NULL, ::GetCurrentThreadId());
   ...
}
```

WH_CBT是众多hook 类型中的一种，意味着安装一个Computer-Based Training（CBT）滤网函数。安装之后，Windows 系统在进行以下任何一个动作之前，会先调用你的滤网函数：

- 令一个窗口成为作用中的窗口（HCBT_ACTIVATE）
- 产生或摧毁一个窗口（HCBT_CREATEWND、HCBT_DESTROYWND）

- 最大化或最小化一个窗口（HCBT_MINMAX）
- 搬移或缩放一个窗口（HCBT_MOVESIZE）
- 完成一个来自系统选单的系统命令（HCBT_SYSTEMCOMMAND）
- 从系统伫列中移去一个滑鼠或键盘消息（HCBT_KEYSKIPPED、HCBT_CLICKSKIPPED）

因此，经过上述hook 安装之后，任何窗口即将产生之前，滤网函数 _AfxCbtFilterHook一定会先被调用：

```
_AfxCbtFilterHook(int code, WPARAM wParam, LPARAM lParam)
{
    _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
    if (code != HCBT_CREATEWND)
    {
            // wait for HCBT_CREATEWND just pass others on...
            return CallNextHookEx(pThreadState->m_hHookOldCbtFilter, code,
                    wParam, lParam);
    }
    ...
        if (!afxData.bWin31)
        {
                // perform subclassing right away on Win32
                _AfxStandardSubclass((HWND)wParam);
        }
        else
        {
            ...
        }
    ...
    LRESULT lResult = CallNextHookEx(pThreadState->m_hHookOldCbtFilter, code,
            wParam, lParam);
    return lResult;
}

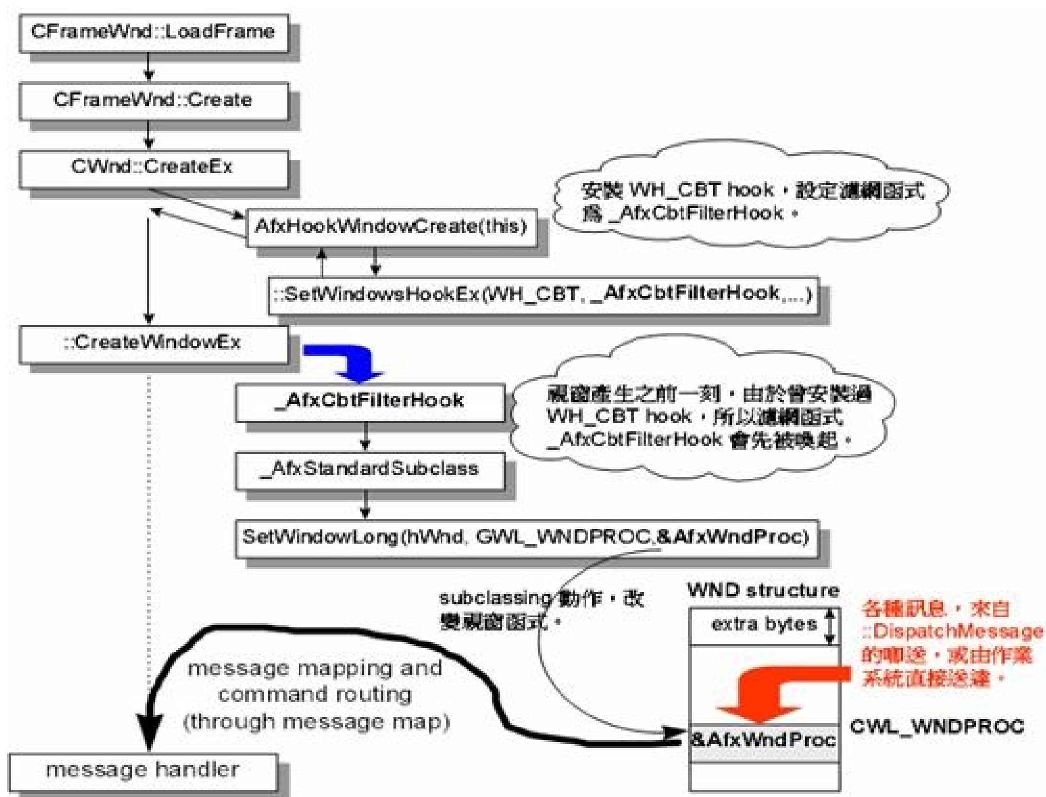
void AFXAPI _AfxStandardSubclass(HWND hWnd)
{
    ...
    // subclass the window with standard AfxWndProc
    oldWndProc = (WNDPROC)SetWindowLong(hWnd, GWL_WNDPROC,
            (DWORD)AfxGetAfxWndProc());
}

WNDPROC AFXAPI AfxGetAfxWndProc()
{

    ...
    return &AfxWndProc;
}
```

啊，非常明显，上面的函数合力做了偷天换日的勾当：把「窗口所属之 Windows 窗口类」中所记录的窗口函数，改换为AfxWndProc。于是，::DispatchMessage 就把消息源源推往AfxWndProc 去了。

这种看起来很迂回又怪异的作法，是为了包容新的3D Controls（细节就容我省略了吧），并与MFC 2.5相容。下图把前述的hook和subclassing 动作以流程图显示出来：



不能稍息，我们还没有走出迷宫！AfxWndProc 只是消息两万五千里长征的第一站！

两万五千里长征——消息的流窜

一个消息从发生到被攫取，直至走向它的归宿，是一条漫漫长路。上一节我们来到了漫漫长路的起头 AfxWndProc，这一节我要带你看看消息实际上如何推动。

消息的流动路线已隐隐有脉络可寻，此脉络是指由BEGIN_MESSAGE_MAP和END_MESSAGE_MAP 以及许许多多ON_WM_xxx 宏所构成的消息映射网。但是唧筒与方向盘是如何设计的？一切的线索还是要靠原始代码透露：

```
// in WINCORE.CPP (MFC 4.x)
LRESULT CALLBACK AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    ...
    // messages route through message map
    CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);
    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
}
LRESULT AFXAPI AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
WPARAM wParam = 0, LPARAM lParam = 0)
{
    ...
    // delegate to object's WindowProc
    lResult = pWnd->WindowProc(nMsg, wParam, lParam);
    ...
    return lResult;
}
```

整个MFC中，拥有虚函数WindowProc者包括CWnd、CControlBar、COleControl、COlePropertyPage、CDialog、CReflectorWnd、CParkingWnd。一般窗口（例如 Frame 窗口、View 窗口）都派生自CWnd，所以让我们看看 CWnd::WindowProc。这个函数相当于C++中的窗口函数：

```
// in WINCORE.CPP (MFC 4.x)
LRESULT CWnd::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
// OnWndMsg does most of the work, except for DefWindowProc call
    LRESULT lResult = 0;
    if (!OnWndMsg(message, wParam, lParam, &lResult))
        lResult = DefWindowProc(message, wParam, lParam);
    return lResult;
}
LRESULT CWnd::DefWindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    if (m_pfnSuper != NULL)
        return::CallWindowProc(m_pfnSuper,m_hWnd,nMsg,wParam,lParam);
    WNDPROC pfnWndProc;
    if ((pfnWndProc = *GetSuperWndProcAddr()) == NULL)
        return::DefWindowProc(m_hWnd,nMsg,wParam,lParam);
    else
        return::CallWindowProc(pfnWndProc,m_hWnd,nMsg,wParam,lParam);
}
```

直线上溯（一般 Windows 消息）

CWnd::WindowProc 调用的OnWndMsg是用来分辨并处理消息的专职机构；如果是命令消息，就交给OnCommand处理，如果是通告消息（Notification），就交给OnNotify处理。WM_ACTIVATE和WM_SETCURSOR也都有特定的处理函数。而一般的Windows消息，就直接在消息映射表中上溯，寻找其归宿（消息处理例程）。为什么要特别区隔出命令消息WM_COMMAND和通告消息WM_NOTIFY两类呢？因为它们的上溯路径不是那么单纯地只往父类去，它们可能需要拐个弯。

```
#0001  // in WINCORE.CPP (MFC 4.0)
#0002  BOOL CWnd::OnWndMsg(UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult)
#0003  {
#0004      LRESULT lResult = 0;
#0005
#0006      // special case for commands
#0007      if (message == WM_COMMAND)
#0008      {
#0009          OnCommand(wParam, lParam);
#0010          ...
#0011      }
#0012
#0013      // special case for notifies
#0014      if (message == WM_NOTIFY)
#0015      {
#0016          OnNotify(wParam, lParam, &lResult);
#0017          ...
#0018      }
#0019      ...
#0020      const AFX_MSGMAP* pMessageMap; pMessageMap = GetMessageMap();
#0021      UINT iHash; iHash = (LOWORD((DWORD)pMessageMap) ^ message) & (iHashMax-1);
#0022      AfxLockGlobals(CRIT_WINMSGCACHE);
#0023      AFX_MSG_CACHE msgCache; msgCache = _afxMsgCache[iHash];
#0024      AfxUnlockGlobals(CRIT_WINMSGCACHE);
#0025
#0026      const AFX_MSGMAP_ENTRY* lpEntry;
#0027      if (...)  // 檢查是否在 chche 之中
#0028      {
#0029          // cache hit
#0030          lpEntry = msgCache.lpEntry;
#0031          if (lpEntry == NULL)
#0032                  return FALSE;
#0033
#0034          // cache hit, and it needs to be handled
#0035          if (message < 0xC000)
#0036                  goto LDispatch;
#0037          else
#0038                  goto LDispatchRegistered;
#0039      }
#0040      else
#0041      {
#0042          // not in cache, look for it
#0043          msgCache.nMsg = message;
#0044          msgCache.pMessageMap = pMessageMap;
#0045
#0046          for (/* pMessageMap already init'ed */; pMessageMap != NULL;
#0047                   pMessageMap = pMessageMap->pBaseMap)
#0048          {
#0049              // 利用 AfxFindMessageEntry 尋找訊息映射表中
#0050              // 對應的訊息處理常式。如果找到，再依 nMsg 爲一般訊息
#0051              // (< 0xC000) 或自行註冊之訊息 (> 0xC000) 分別跳到
#0052              // LDispatch: 或 LDispatchRegistered: 去執行。
#0053
#0054              // Note: catch not so common but fatal mistake!!
#0055              //      BEGIN_MESSAGE_MAP(CMyWnd, CMyWnd)
#0056
#0057              if (message < 0xC000)
#0058              {
```

```
#0059                // constant window message
#0060                if ((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,
#0061                    message, 0, 0)) != NULL)
#0062                {
#0063                    msgCache.lpEntry = lpEntry;
#0064                    goto LDispatch;
#0065                }
#0066            }
#0067            else
#0068            {
#0069                // registered windows message
#0070                lpEntry = pMessageMap->lpEntries;
#0071                while ((lpEntry = AfxFindMessageEntry(lpEntry, 0xC000, 0, 0))
#0072                    != NULL)
#0073                {
#0074                    UINT* pnID = (UINT*)(lpEntry->nSig);
#0075                    ASSERT(*pnID >= 0xC000);
#0076                            // must be successfully registered
#0077                    if (*pnID == message)
#0078                    {
#0079                            msgCache.lpEntry = lpEntry;
#0080                            goto LDispatchRegistered;
#0081                    }
#0082                    lpEntry++;      // keep looking past this one
#0083                }
#0084            }
#0085        }
#0086        msgCache.lpEntry = NULL;
#0087        return FALSE;
#0088    }
#0089    ASSERT(FALSE);      // not reached
#0090
#0091 LDispatch:
#0092    union MessageMapFunctions mmf;
#0093    mmf.pfn = lpEntry->pfn;
#0094
#0095    switch (lpEntry->nSig)
#0096    {
#0097    case AfxSig_bD:
#0098            lResult = (this->*mmf.pfn_bD)(CDC::FromHandle((HDC)wParam));
#0099            break;
#0100
#0101    case AfxSig_bb:     // AfxSig_bb, AfxSig_bw, AfxSig_bh
#0102            lResult = (this->*mmf.pfn_bb)((BOOL)wParam);
#0103            break;
#0104
#0105    case AfxSig_bWww:   // really AfxSig_bWiw
#0106            lResult = (this->*mmf.pfn_bWww)(CWnd::FromHandle((HWND)wParam),
#0107                (short)LOWORD(lParam), HIWORD(lParam));
#0108            break;
#0109
#0110    case AfxSig_bHELPINFO:
#0111            lResult = (this->*mmf.pfn_bHELPINFO)((HELPINFO*)lParam);
#0112            break;
#0113
#0114    case AfxSig_is:
#0115            lResult = (this->*mmf.pfn_is)((LPTSTR)lParam);
#0116            break;
#0117
#0118    case AfxSig_lwl:
```

```
#0119             lResult = (this->*mmf.pfn_lwl)(wParam, lParam);
#0120             break;
#0121
#0122     case AfxSig_vv:
#0123             (this->*mmf.pfn_vv)();
#0124             break;
#0125     ...
#0126     }
#0127     goto LReturnTrue;
#0128
#0129 LDispatchRegistered:    // for registered windows messages
#0130     ASSERT(message >= 0xC000);
#0131     mmf.pfn = lpEntry->pfn;
#0132     lResult = (this->*mmf.pfn_lwl)(wParam, lParam);
#0133
#0134 LReturnTrue:
#0135     if (pResult != NULL)
#0136             *pResult = lResult;
#0137     return TRUE;
#0138 }
```

```
#0001 AfxFindMessageEntry(const AFX_MSGMAP_ENTRY* lpEntry,
#0002         UINT nMsg, UINT nCode, UINT nID)
#0003 {
#0004 #if defined(_M_IX86) && !defined(_AFX_PORTABLE)
#0005 // 32-bit Intel 386/486 version.
#0006         ... // 以組合語言碼處理，加快速度。
#0007 #else  // _AFX_PORTABLE
#0008         // C version of search routine
#0009         while (lpEntry->nSig != AfxSig_end)
#0010         {
#0011             if (lpEntry->nMessage == nMsg && lpEntry->nCode == nCode &&
#0012                 nID >= lpEntry->nID && nID <= lpEntry->nLastID)
#0013             {
#0014                 return lpEntry;
#0015             }
#0016             lpEntry++;
#0017         }
#0018         return NULL;    // not found
#0019 #endif  // _AFX_PORTABLE
#0020 }
```

直线上溯的逻辑实在是相当单纯的了，唯一做的动作就是比对消息映射表，如果吻合就调用表中项目所记录的函数。比对的对象有二，一个是原原本本的消息映射表（那个巨大的结构），另一个是MFC为求快速所设计的一个cache（cache 的实现太过复杂，我并没有把它的原始代码表现出来）。比对成功后，调用对应之函数时，有一个巨大的 switch/case动作，那是为了确保类型安全（type-safe）。稍后我有一个小节详细讨论之。

拐弯上溯（WM_COMMAND 命令消息）

如果消息是WM_COMMAND，你看到了，CWnd::OnWndMsg（上节所述）另辟蹊跷，交由OnCommand来处理。这并不一定就指的是CWnd::OnCommand，得视this 指针指向哪一种对象而定。在MFC之中，以下数个类都改写了OnCommand虚函数：

```
class CWnd : public CCmdTarget
class CFrameWnd : public CWnd
class CMDIFrameWnd : public CFrameWnd
class CSplitterWnd : public CWnd
class CPropertySheet : public CWnd
class COlePropertyPage : public CDialog
```

我们挑一个例子来看。假设消息是从 CFrameWnd 进来的好了，于是：

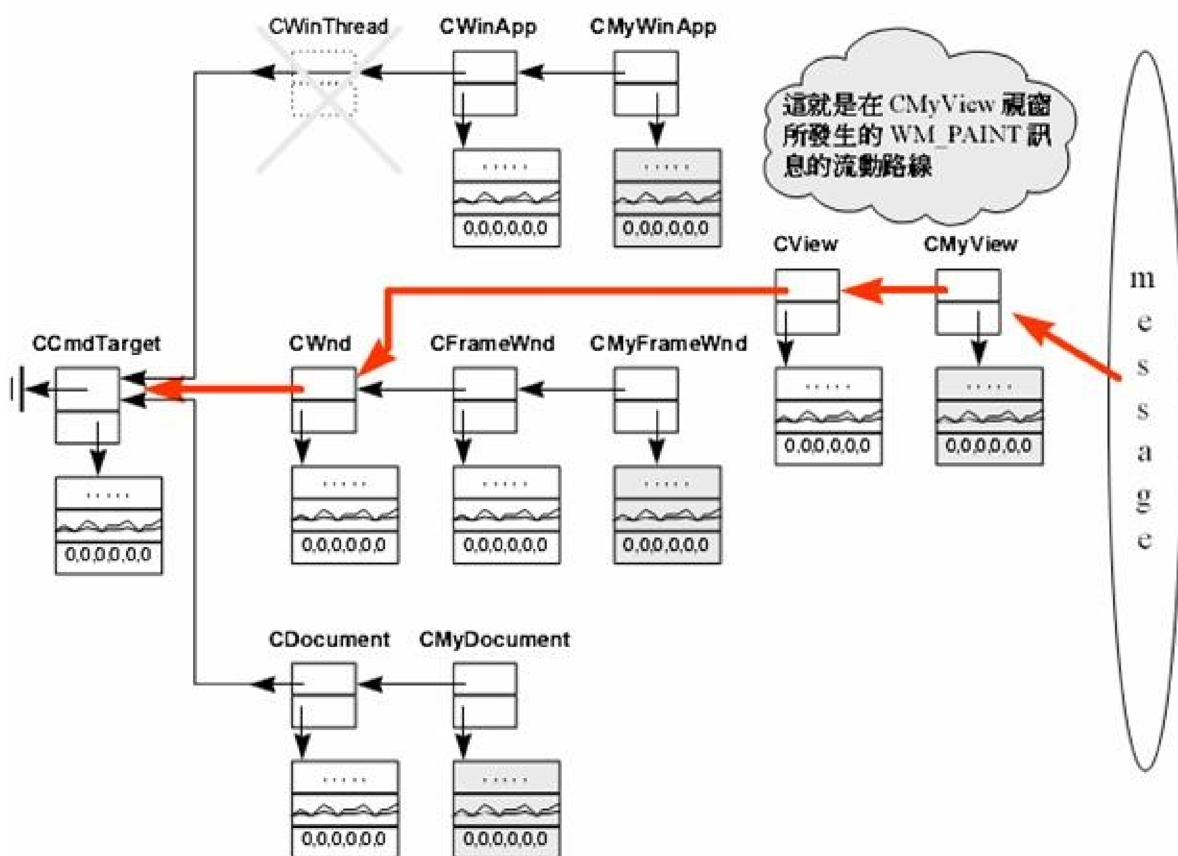


图9-3 当WM_PAINT发生于View窗口，消息的流动路线

```
// in FRMWND.CPP (MFC 4.0)
BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    ...
    // route as normal command
    return CWnd::OnCommand(wParam, lParam);
}
// in WINCORE.CPP (MFC 4.0)
BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    ...
    return OnCmdMsg(nID, nCode, NULL, NULL);
}
```

这里调用的OnCmdMsg 并不一定就是指CWnd::OnCmdMsg，得看this指针指向哪一种对象而定。目前情况是指向一个CFrameWnd对象，而MFC之中「拥有」OnCmdMsg的类（注意，此话有语病，我应该说MFC之中「曾经改写」过OnCmdMsg 的类）是：

```
class CCmdTarget : public CObject
class CFrameWnd : public CWnd
class CMDIFrameWnd : public CFrameWnd
class CView : public CWnd
class CPropertySheet : public CWnd
class CDialog : public CWnd
class CDocument : public CCmdTarget
class COleDocument : public CDocument
```

显然我们应该往 CFrameWnd 追踪：

```
// in FRMWND.CPP (MFC 4.0)
BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo)
{
        // pump through current view FIRST
        CView* pView = GetActiveView();
        if (pView != NULL && pView->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        // then pump through frame
        if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        // last but not least, pump through app
        CWinApp* pApp = AfxGetApp();
        if (pApp != NULL && pApp->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        return FALSE;
}
```

这里非常明显地兵分三路，正是为了实践MFC这个Application Framework 对于命令消息的绕行路线的规划：



图9-4 MFC 对于命令消息WM_COMMAND的特殊处理顺序

让我们锲而不舍地追踪下去：

```
// in VIEWCORE.CPP (MFC 4.0)
BOOL CView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo)
{
    // first pump through pane
    if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    // then pump through document
    BOOL bHandled = FALSE;
    if (m_pDocument != NULL)
    {
        // special state for saving view before routing to document
        _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
        CView* pOldRoutingView = pThreadState->m_pRoutingView;
        pThreadState->m_pRoutingView = this;
        bHandled = m_pDocument->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
        pThreadState->m_pRoutingView = pOldRoutingView;
    }

    return bHandled;
}
```

这反应出图9-4 搜寻路径中「先View 而后Document」的规划。由于CWnd并未改写OnCmdMsg，所以函数中调用的CWnd::OnCmdMsg，其实就是 CCmdTarget::OnCmdMsg：

```
// in CMDTARG.CPP (MFC 4.0)
BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode, void* pExtra,
       AFX_CMDHANDLERINFO* pHandlerInfo)
{
    ...
    // look through message map to see if it applies to us
    for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
         pMessageMap = pMessageMap->pBaseMap)
    {
        lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries, nMsg, nCode, nID);
        if (lpEntry != NULL)
        {
            // found it
            return DispatchCmdMsg(this, nID, nCode,
                    lpEntry->pfn, pExtra, lpEntry->nSig, pHandlerInfo);
        }
    }
    return FALSE;   // not handled
}
```

其中的AfxFindMessageEntry动作稍早我已列出。

当命令消息兵分三路的第一路走到消息映射网的末尾一个类 CCmdTarget，没有办法再「节外生枝」，只能乖乖比对CCmdTarget的消息映射表。如果没有发现吻合者，传回FALSE，引起CView::OnCmdMsg接下去调用 m_pDocument->OnCmdMsg。如果有吻合者，调用全局函数DispatchCmdMsg：

```
static BOOL DispatchCmdMsg(CCmdTarget* pTarget, UINT nID, int nCode,
        AFX_PMSG pfn, void* pExtra, UINT nSig, AFX_CMDHANDLERINFO* pHandlerInfo)
                // return TRUE to stop routing
{
        ASSERT_VALID(pTarget);
        UNUSED(nCode);   // unused in release builds

        union MessageMapFunctions mmf;
        mmf.pfn = pfn;
        BOOL bResult = TRUE; // default is ok
        ...
        switch (nSig)
        {
        case AfxSig_vv:
                // normal command or control notification
                (pTarget->*mmf.pfn_COMMAND)();
                break;

        case AfxSig_bv:
                // normal command or control notification
                bResult = (pTarget->*mmf.pfn_bCOMMAND)();
                break;

        case AfxSig_vw:
                // normal command or control notification in a range
                (pTarget->*mmf.pfn_COMMAND_RANGE)(nID);
                break;

        case AfxSig_bw:
                // extended command (passed ID, returns bContinue)
                bResult = (pTarget->*mmf.pfn_COMMAND_EX)(nID);
                break;
        ...
        default:    // illegal
                ASSERT(FALSE);
                return 0;
        }
        return bResult;
}
```

以下是另一路 CDocument 的动作：

```
// in DOCCORE.CPP
BOOL CDocument::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo)
{
        if (CCmdTarget::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        // otherwise check template
        if (m_pDocTemplate != NULL &&
          m_pDocTemplate->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        return FALSE;
}
```

图9-5画出FrameWnd窗口收到命令消息后的四个尝试路径。第 3 章曾经以一个简单的DOS 程序仿真出这样的绕行路线。

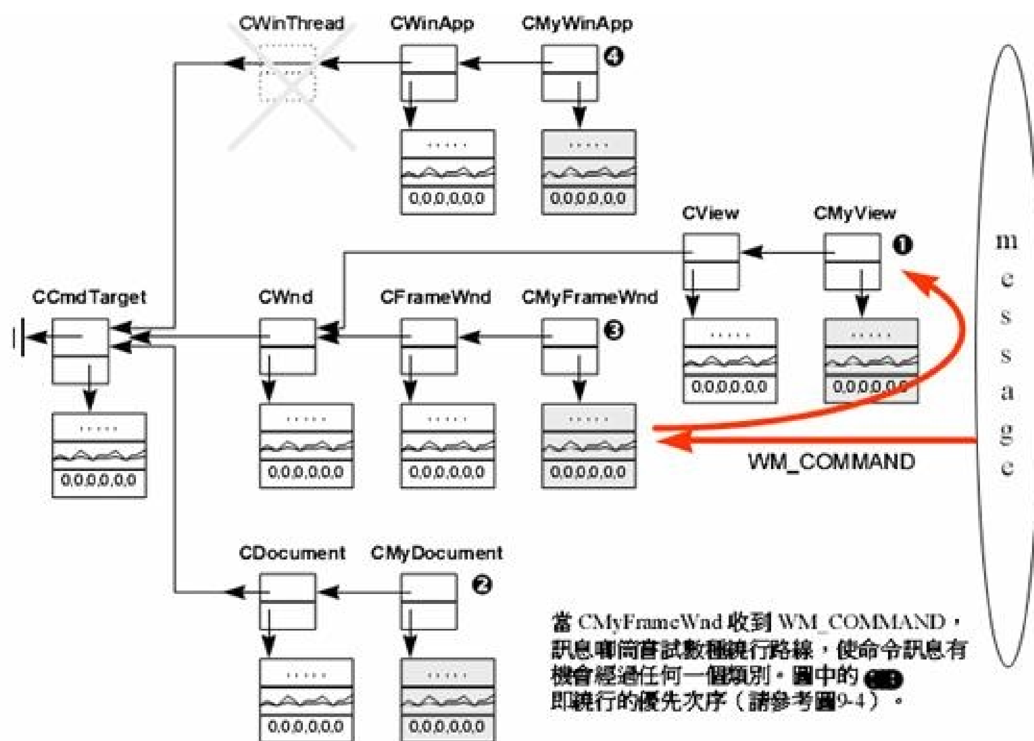


图9-5 FrameWnd 窗口收到命令消息后的四个尝试路径。

第 3 章曾经以一个 简 单的DOS 程序仿真出这样的绕行路线。

OnCmdMsg是各类专门用来对付命令消息的函数。每一个「可接受命令消息之对象」（Command Target）在处理命令消息时都会（都应该）遵循一个游戏规则：调用另一个目标类的OnCmdMsg。这才能够将命令消息传送下去。如果说AfxWndProc是消息流动的「唧筒」，各类的OnCmdMsg 就是消息流动的「转辙器」。

以下我举一个具体例子。假设命令消息从Scribble 的【Edit/Clear All】发出，其处理常式位在CScribbleDoc，下面是这个命令消息的流浪过程：

1.MDI主 视 窗 （ CMDIFrameWnd）收到命令消息WM_COMMAND，其ID为ID_EDIT_CLEAR_ALL。

2.MDI主窗口把命令消息交给目前作用中的MDI子窗口（CMDIChildWnd）。

3.MDI子窗口给它自己的子窗口（也就是View）一个机会。

4.View检查自己的Message Map。

5.View发现没有任何处理例程可以处理此命令消息，只好把它传给 Document。

6.Document检查自己的Message Map，它发现了一个吻合项：

```
BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
...
END_MESSAGE_MAP()
```

于是调用该函数，命令消息的流动路线也告终止。

如果上述的步骤 6 仍没有找到处理函数，那么就：

7.Document 把这个命令消息再送到Document Template 对象去。

8.还是没被处理，于是命令消息回到View。

9.View 没有处理，于是又回给MDI子窗口本身。

10.传给CWinApp 对象——无主消息的终极归属。

图9-6 是构成「消息捕获」之各个函数的调用次序。此图可以对前面所列之各个原始代码组织出一个大局观来。

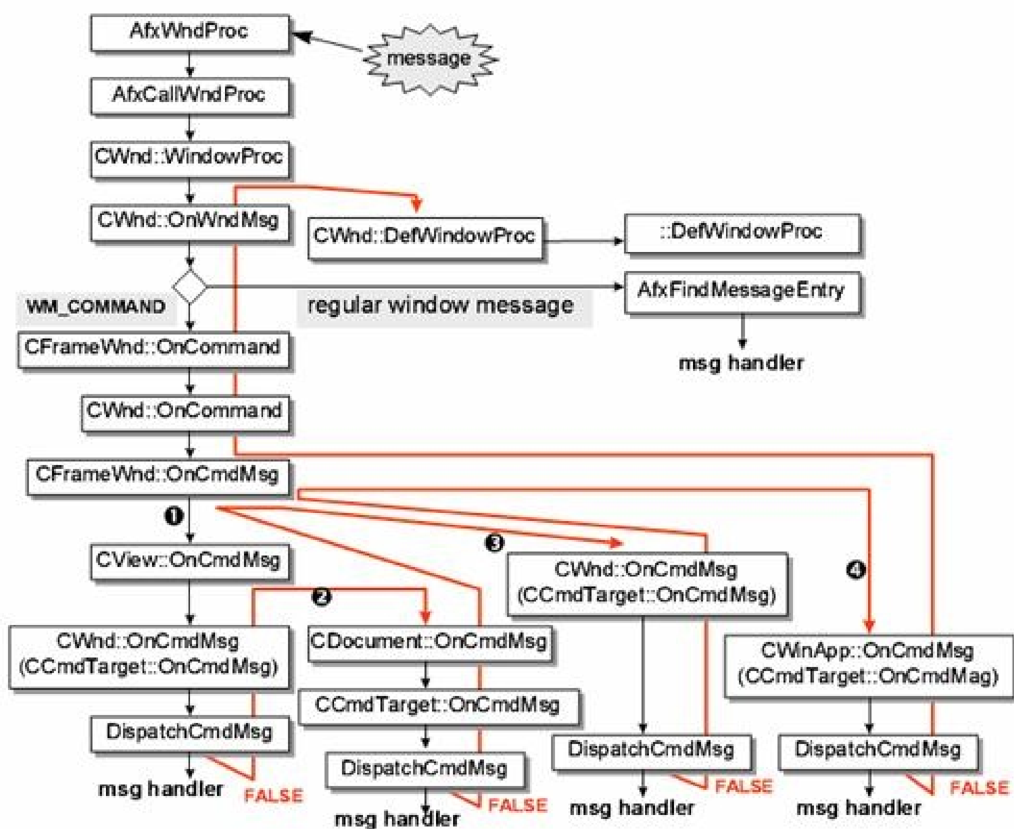


图9-6 构成「消息捕获」之各个函数的调用次序

罗塞达碑石：AfxSig_xx 的奥秘

大架构建立起来了，但我还没有很仔细地解释在消息映射「网」中的 _messageEntries[]数组内容。为什么消息经由推动引擎（上一节谈的那整套家伙）推过这些数组，就可以找到它的处理例程？

Paul DiLascia 在他的文章（¨Meandering Through the Maze of MFC Message and Command Routing¨，Microsoft Systems Journal，1995/07）中形容这些数组之内一笔一笔的记录像是罗塞达碑石，呵呵，就靠它们揭开消息映射的最后谜底了。

罗塞达碑石（Rosetta Stone），1799 年拿破仑远征埃及时，由一名官员在尼罗河口罗塞达发现，揭开了古埃及象形文字之谜。石碑是黑色玄武岩，高 114 公分，厚 28 公分，宽72 公分。经法国学者 Jean-Francois Champollion 研究后，世人因得顺利研读古埃及文献。

消息映射表的每一笔记录是这样的形式：

```
struct AFX_MSGMAP_ENTRY
{
    UINT nMessage; // windows message
    UINT nCode; // control code or WM_NOTIFY code
    UINT nID; // control ID (or 0 for windows messages)
    UINT nLastID; // used for entries specifying a range of control id's
    UINT nSig; // signature type (action) or pointer to message #
    AFX_PMSG pfn; // routine to call (or special value)
};
```

内中包括一个Windows 消息、其控制组件ID以及通告代码（notification code，对消息的更多描述，例如EN*CHANGED*或*CBN_DROPDIOWN* 等）、一个签名记号、以及一个*CCmdTarget* 派生类的成员函数。任何一个*ON* 宏会把这六个项目初始化起来。例如：

```
#define ON_WM_CREATE() \
 { WM_CREATE, 0, 0, 0, AfxSig_is, \
   (AFX_PMSG)(AFX_PMSGW)(int (AFX_MSG_CALL CWnd::*)(LPCREATESTRUCT))OnCreate },
```

你看到了可怕的类型转换动作，这完全是为了保持类型安全（type-safe）。

有一个很莫名其妙的东西：AfxSig_。要了解它作什么用，你得先停下来几分钟，想想另一个问题：当上一节的推动引擎比对消息并发现吻合之后，就调用对应的处理例程，但它怎么知道要交给消息处理例程哪些参数呢？要知道，不同的消息处理例程需要不同的参数（包括个数和类型），而其函数指针（AFX_PMSG）却都被定义为这付德行：

```
typedef void (AFX_MSG_CALL CCmdTarget::*AFX_PMSG)(void);
```

这么简陋的信息无法表现应该传递什么样的参数，而这正是 AfxSig_ 要贡献的地方。当推动引擎比对完成，欲调用某个消息处理例程 lpEntry->pfn 时，动作是这样子地（出现在 CWnd::OnWndMsg 和 DispatchCmdMsg 中）：

```
union MessageMapFunctions mmf;
mmf.pfn = lpEntry->pfn;
switch (lpEntry->nSig)
{
    case AfxSig_is:
        lResult = (this->*mmf.pfn_is)((LPTSTR)lParam);
        break;
    case AfxSig_lwl:
        lResult = (this->*mmf.pfn_lwl)(wParam, lParam);
        break;
    case AfxSig_vv:
        (this->*mmf.pfn_vv)();
        break;
    ...
}
```

注意两样东西：MessageMapFunctions和AfxSig。*AfxSig* 定义于 AFXMSG_.H 檔：

```
enum AfxSig
{
       AfxSig_end = 0,     // [marks end of message map]

       AfxSig_bD,      // BOOL (CDC*)
       AfxSig_bb,      // BOOL (BOOL)
       AfxSig_bWww,    // BOOL (CWnd*, UINT, UINT)
       AfxSig_hDWw,    // HBRUSH (CDC*, CWnd*, UINT)
       AfxSig_hDw,     // HBRUSH (CDC*, UINT)

AfxSig_iwWw,    // int (UINT, CWnd*, UINT)
AfxSig_iww,     // int (UINT, UINT)
AfxSig_iWww,    // int (CWnd*, UINT, UINT)
AfxSig_is,      // int (LPTSTR)
AfxSig_lwl,     // LRESULT (WPARAM, LPARAM)
AfxSig_lwwM,    // LRESULT (UINT, UINT, CMenu*)
AfxSig_vv,      // void (void)

AfxSig_vw,      // void (UINT)
AfxSig_vww,     // void (UINT, UINT)
AfxSig_vvii,    // void (int, int) // wParam is ignored
AfxSig_vwww,    // void (UINT, UINT, UINT)
AfxSig_vwii,    // void (UINT, int, int)
AfxSig_vwl,     // void (UINT, LPARAM)
AfxSig_vbWW,    // void (BOOL, CWnd*, CWnd*)
AfxSig_vD,      // void (CDC*)
AfxSig_vM,      // void (CMenu*)
AfxSig_vMwb,    // void (CMenu*, UINT, BOOL)

AfxSig_vW,      // void (CWnd*)
AfxSig_vWww,    // void (CWnd*, UINT, UINT)
AfxSig_vWp,     // void (CWnd*, CPoint)
AfxSig_vWh,     // void (CWnd*, HANDLE)
AfxSig_vwW,     // void (UINT, CWnd*)
AfxSig_vwWb,    // void (UINT, CWnd*, BOOL)
AfxSig_vwwW,    // void (UINT, UINT, CWnd*)
AfxSig_vwwx,    // void (UINT, UINT)
AfxSig_vs,      // void (LPTSTR)
AfxSig_vOWNER,  // void (int, LPTSTR), force return TRUE
AfxSig_iis,     // int (int, LPTSTR)
AfxSig_wp,      // UINT (CPoint)
AfxSig_wv,      // UINT (void)
AfxSig_vPOS,    // void (WINDOWPOS*)
AfxSig_vCALC,   // void (BOOL, NCCALCSIZE_PARAMS*)
AfxSig_vNMHDRpl,    // void (NMHDR*, LRESULT*)
AfxSig_bNMHDRpl,    // BOOL (NMHDR*, LRESULT*)

AfxSig_vwNMHDRpl,   // void (UINT, NMHDR*, LRESULT*)
AfxSig_bwNMHDRpl,   // BOOL (UINT, NMHDR*, LRESULT*)
AfxSig_bHELPINFO,   // BOOL (HELPINFO*)
AfxSig_vwSIZING,    // void (UINT, LPRECT) -- return TRUE

// signatures specific to CCmdTarget
AfxSig_cmdui,   // void (CCmdUI*)
AfxSig_cmduiw,  // void (CCmdUI*, UINT)
AfxSig_vpv,     // void (void*)
AfxSig_bpv,     // BOOL (void*)
```

```
        // Other aliases (based on implementation)
        AfxSig_vwwh,              // void (UINT, UINT, HANDLE)
        AfxSig_vwp,               // void (UINT, CPoint)
        AfxSig_bw = AfxSig_bb,     // BOOL (UINT)
        AfxSig_bh = AfxSig_bb,     // BOOL (HANDLE)
        AfxSig_iw = AfxSig_bb,     // int (UINT)
        AfxSig_ww = AfxSig_bb,     // UINT (UINT)
        AfxSig_bv = AfxSig_wv,     // BOOL (void)
        AfxSig_hv = AfxSig_wv,     // HANDLE (void)
        AfxSig_vb = AfxSig_vw,     // void (BOOL)
        AfxSig_vbh = AfxSig_vww,    // void (BOOL, HANDLE)
        AfxSig_vbw = AfxSig_vww,    // void (BOOL, UINT)
        AfxSig_vhh = AfxSig_vww,    // void (HANDLE, HANDLE)
        AfxSig_vh = AfxSig_vw,     // void (HANDLE)
        AfxSig_viSS = AfxSig_vwl,   // void (int, STYLESTRUCT*)
        AfxSig_bwl = AfxSig_lwl,
        AfxSig_vwMOVING = AfxSig_vwSIZING,  // void (UINT, LPRECT) -- return TRUE
};
```

MessageMapFunctions 定义于 WINCORE.CPP檔：

```
union MessageMapFunctions
{
        AFX_PMSG pfn;   // generic member function pointer

        // specific type safe variants
        BOOL    (AFX_MSG_CALL CWnd::*pfn_bD)(CDC*);
        BOOL    (AFX_MSG_CALL CWnd::*pfn_bb)(BOOL);
        BOOL    (AFX_MSG_CALL CWnd::*pfn_bWww)(CWnd*, UINT, UINT);
        BOOL    (AFX_MSG_CALL CWnd::*pfn_bHELPINFO)(HELPINFO*);
        HBRUSH  (AFX_MSG_CALL CWnd::*pfn_hDWw)(CDC*, CWnd*, UINT);
        HBRUSH  (AFX_MSG_CALL CWnd::*pfn_hDw)(CDC*, UINT);
        int     (AFX_MSG_CALL CWnd::*pfn_iwWw)(UINT, CWnd*, UINT);
        int     (AFX_MSG_CALL CWnd::*pfn_iww)(UINT, UINT);
        int     (AFX_MSG_CALL CWnd::*pfn_iWww)(CWnd*, UINT, UINT);
        int     (AFX_MSG_CALL CWnd::*pfn_is)(LPTSTR);
        LRESULT (AFX_MSG_CALL CWnd::*pfn_lwl)(WPARAM, LPARAM);
        LRESULT (AFX_MSG_CALL CWnd::*pfn_lwwM)(UINT, UINT, CMenu*);
        void    (AFX_MSG_CALL CWnd::*pfn_vv)(void);

        void    (AFX_MSG_CALL CWnd::*pfn_vw)(UINT);
        void    (AFX_MSG_CALL CWnd::*pfn_vww)(UINT, UINT);
        void    (AFX_MSG_CALL CWnd::*pfn_vvii)(int, int);
        void    (AFX_MSG_CALL CWnd::*pfn_vwww)(UINT, UINT, UINT);
        void    (AFX_MSG_CALL CWnd::*pfn_vwii)(UINT, int, int);
```

```
        void    (AFX_MSG_CALL CWnd::*pfn_vwl)(WPARAM, LPARAM);
        void    (AFX_MSG_CALL CWnd::*pfn_vbWW)(BOOL, CWnd*, CWnd*);
        void    (AFX_MSG_CALL CWnd::*pfn_vD)(CDC*);
        void    (AFX_MSG_CALL CWnd::*pfn_vM)(CMenu*);
        void    (AFX_MSG_CALL CWnd::*pfn_vMwb)(CMenu*, UINT, BOOL);

        void    (AFX_MSG_CALL CWnd::*pfn_vW)(CWnd*);
        void    (AFX_MSG_CALL CWnd::*pfn_vWww)(CWnd*, UINT, UINT);
        void    (AFX_MSG_CALL CWnd::*pfn_vWp)(CWnd*, CPoint);
        void    (AFX_MSG_CALL CWnd::*pfn_vWh)(CWnd*, HANDLE);
        void    (AFX_MSG_CALL CWnd::*pfn_vwW)(UINT, CWnd*);
        void    (AFX_MSG_CALL CWnd::*pfn_vwWb)(UINT, CWnd*, BOOL);
        void    (AFX_MSG_CALL CWnd::*pfn_vwwW)(UINT, UINT, CWnd*);
        void    (AFX_MSG_CALL CWnd::*pfn_vwwx)(UINT, UINT);
        void    (AFX_MSG_CALL CWnd::*pfn_vs)(LPTSTR);
        void    (AFX_MSG_CALL CWnd::*pfn_vOWNER)(int, LPTSTR);   // force return TRUE
        int     (AFX_MSG_CALL CWnd::*pfn_iis)(int, LPTSTR);
        UINT    (AFX_MSG_CALL CWnd::*pfn_wp)(CPoint);
        UINT    (AFX_MSG_CALL CWnd::*pfn_wv)(void);
        void    (AFX_MSG_CALL CWnd::*pfn_vPOS)(WINDOWPOS*);
        void    (AFX_MSG_CALL CWnd::*pfn_vCALC)(BOOL, NCCALCSIZE_PARAMS*);
        void    (AFX_MSG_CALL CWnd::*pfn_vwp)(UINT, CPoint);
        void    (AFX_MSG_CALL CWnd::*pfn_vwwh)(UINT, UINT, HANDLE);
};
```

其实呢，真正的函数只有一个pfn，但通过union，它有许多态态不同的形象。pfn_vv 代表「参数为void，传回值为void」；pfn_lwl代表「参数为wParam 和lParam，传回值为LRESULT」；pfn_is代表「参数为LPTSTR 字符串，传回值为 int」。

相当精致，但是也有点儿可怖，是不是？使用 MFC 或许应该像吃蜜饯一样；蜜饯很好吃，但你最好不要看到蜜饯的生产过程！唔，我真的不知道！

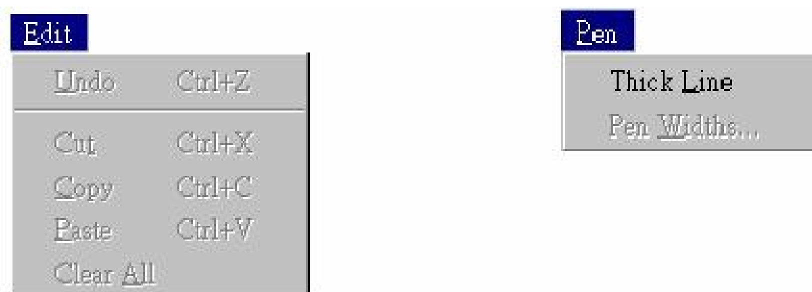
无论如何，我把所有的神秘都揭开在你面前了。

Scribble Step2 ：UI 对象的变化

理论基础建立完毕，该是实现的时候。Step2 将新增三个选单命令项，一个工具列按钮，并维护这些UI对象的使用状态。

改变选单

Step2 将增加一个【Pen】选单，其中有两个命令项目；并在【Edit】选单中增加一个【Clear All】命令项目：



- 【Pen/Thick Line】：这是一个切换开关，允许设定使用粗笔或细笔。如果使用者设定粗笔，我们将在这项目的旁边打个勾（所谓的checked）；如果使用者选择细笔（也就是在打勾的本项目上再按一下），我们就把勾号去除（所谓的unchecked）。
- 【Pen/Pen Widths】：这会唤起一个对话框，允许设定笔的宽度。对话框的设计并不在本章范围，那是下一章的事。
- 【Edit/Clear All】：清除目前作用之Document 数据。当然对应之View窗口内容也应该清干净。

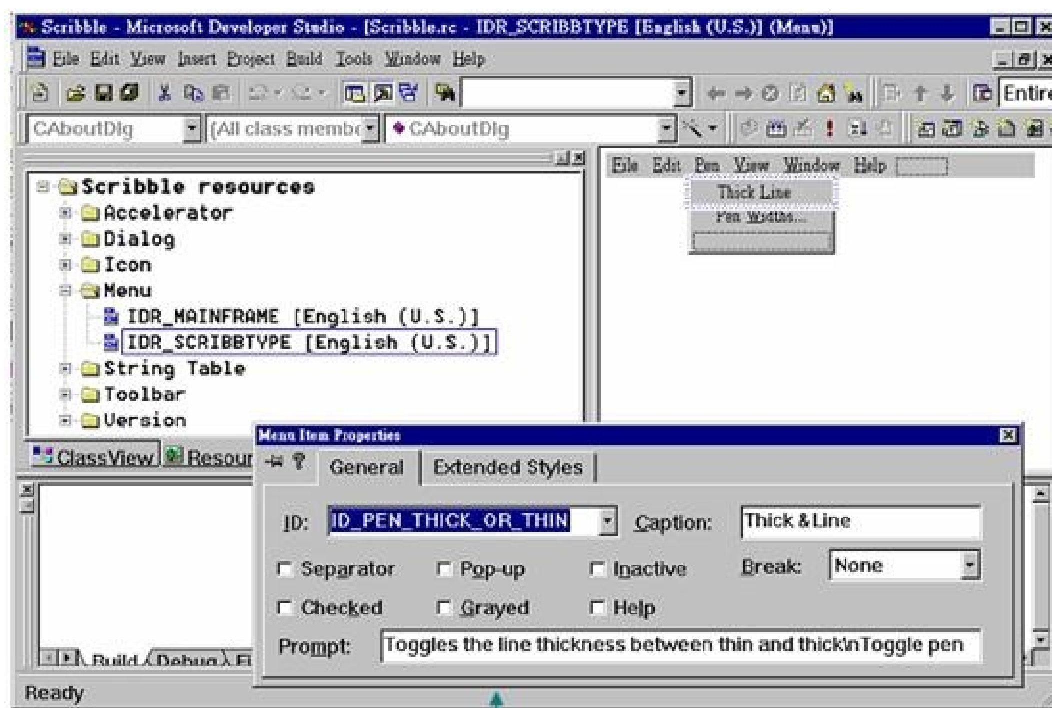
Visual C++ 整合环境中的选单编辑器拥有非常方便的鼠标拖放（drag and drop）功能，所以做出上述的选单命令项不是难事。不过这些命令项目还得经过某些动作，才能与程序代码关联起来发生作用，这方面ClassWizard 可以帮助我们。稍后我会说明这一切。

以下利用Visual C++整合环境中的选单编辑器修改选单：

- 启动选单编辑器（请参考第4 章）。Scribble 有两份选单，IDR_MAINFRAME

适用于没有任何子窗口的情况，IDR_SCRIBTYPE 适用于有子窗口的情况。我

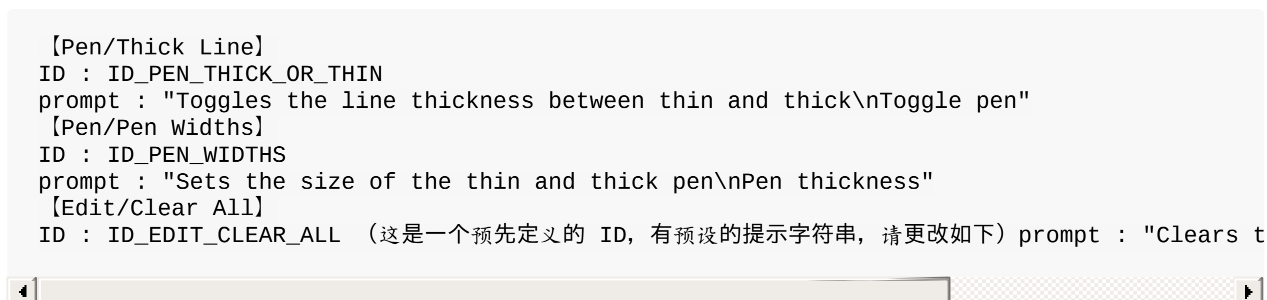
们选择后者。



- IDR_SCRIBTYPE 选单内容出现于画面右半侧。加入新增的三个命令项。每个

命令项会获得一个独一无二的识别代码，定义于 RESOURCE.H 或任何你指定的文件中。图下方的【Menu Item Properties】对话框在你双击某个命令项后出现，允许你更改命令项的识别代码与提示字符串（将出现在状态列中）。如果你对操作过程不熟练，请参考Visual C++ User's Guide（Visual C++ Online 上附有此书之电子版）。

- 三个新命令项的ID 值以及提示字符串整理于下：



```
【Pen/Thick Line】
ID : ID_PEN_THICK_OR_THIN
prompt : "Toggles the line thickness between thin and thick\nToggle pen"
【Pen/Pen Widths】
ID : ID_PEN_WIDTHS
prompt : "Sets the size of the thin and thick pen\nPen thickness"
【Edit/Clear All】
ID : ID_EDIT_CLEAR_ALL （这是一个预先定义的 ID，有预设的提示字符串，请更改如下）prompt : "Clears t
```

注意：每一个提示字符串都有一个\n 子字符串，那是作为工具列按钮的「小黄卷标」的标签内容。「小黄标签」（学名叫作 tool tips）是 Windows 95 新增的功能。

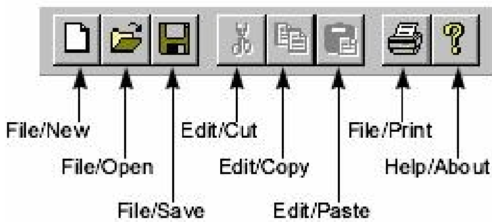
对Framework 而言，命令项的ID是用以识别命令消息的唯一依据。你只需在【Properties】对话框中键入你喜欢的 ID 名称（如果你不满意选单编辑器自动给你的那个），至于它真正的数值不必在意，选单编辑器会在你的 RESOURCE.H 档中加上定义值。

经过上述动作，选单编辑器影响我们的程序代码如下：

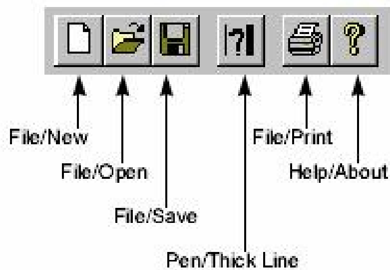
```
// in RESOURCE.H
#define ID_PEN_THICK_OR_THIN 32772
#define ID_PEN_WIDTHS 32773
（注：另一个ID ID_EDIT_CLEAR_ALL已预先定义于AFXRES.H 中）
// in SCRIBBLE.RC
IDR_SCRIBBTYPE MENU PRELOAD DISCARDABLE
BEGIN
...
POPUP "&Edit"
BEGIN
...
MENUITEM "Clear &All", ID_EDIT_CLEAR_ALL
END
POPUP "&Pen"
BEGIN
MENUITEM "Thick &Line", ID_PEN_THICK_OR_THIN
MENUITEM "Pen &Widths...", ID_PEN_WIDTHS
END
...
END
STRINGTABLE DISCARDABLE
BEGIN
ID_PEN_THICK_OR_THIN  "Toggles the line thickness between thin and thick\nToggle pen"
ID_PEN_WIDTHS "Sets the size of the thin and thick pen\nPen thickness"
END
STRINGTABLE DISCARDABLE
BEGIN
ID_EDIT_CLEAR_ALL "Clears the drawing\nErase All"
...
END
```

改变工具列

过去，也就是Visual C++ 4.0之前，改变工具列有点麻烦。你必须先以图形编辑器修改工具列对应之bitmap图形，然后更改程序代码中对应的工具列按钮识别代码。现在可就轻松多了，工具列编辑器让我们一气呵成。主要原因是，工具列现今也成为了资源的一种。下面是 Scribble Step1 的工具列：

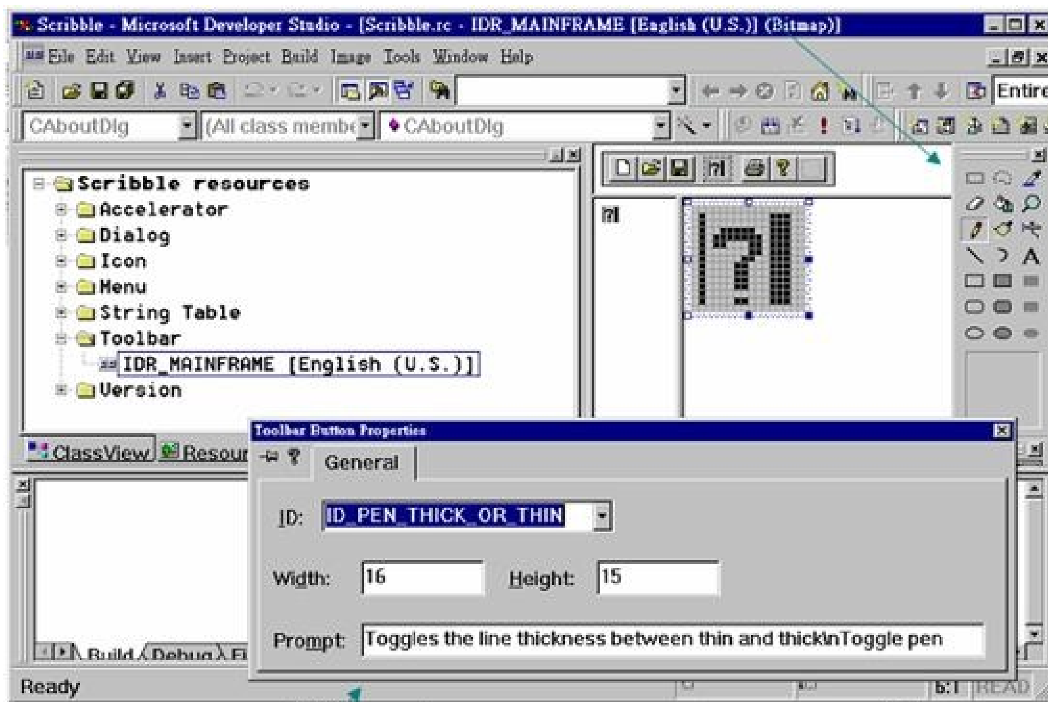


现在我希望为【Pen/Thick Line】命令项设计一个工具列按钮，并且把 Scribble 用不到的三个预设按钮去除（分别是 Cut、Copy、Paste）：



编辑动作如下：

- 启动工具列编辑器，选择IDR_MAINFRAME。有一个绘图工具箱出现在最右侧。



将三个用不着的按钮除去：以鼠标拖拉这些按钮，拉到工具列以外即可。

- 在工具列最右侧的空白按钮上作画，并将它拖拉到适当位置。
- 为了让这个新的按钮起作用，必须指定一个ID给它。我们希望这个按钮相当于【Pen/Thick Line】命令项，所以它的ID当然应该与该命令项的ID相同，也就是ID_PEN_THICK_OR_THIN。双击这个新按钮，出现【Toolbar Button Properties】对话框，请选择正确的ID。注意，由于此一ID 先前已定义好，所以其提示字符串以及小黄卷标也就与此一工具列按钮产生了关联。
- 存文件。
- 工具列编辑器为我们修改了工具列的bitmap图形文件内容：

```
IDR_MAINFRAME  BITMAP  MOVEABLE PURE "res\\Toolbar.bmp"
```

同时，工具列项目也由原来的：

```
IDR_MAINFRAME TOOLBAR DISCARDABLE  16, 15
BEGIN
BUTTON ID_FILE_NEW
BUTTON ID_FILE_OPEN
BUTTON ID_FILE_SAVE
SEPARATOR
BUTTON ID_EDIT_CUT
BUTTON ID_EDIT_COPY
BUTTON ID_EDIT_PASTE
SEPARATOR
BUTTON ID_FILE_PRINT
BUTTON ID_APP_ABOUT
END
```

改变为：

```
IDR_MAINFRAME TOOLBAR DISCARDABLE  16, 15
BEGIN
BUTTON ID_FILE_NEW
BUTTON ID_FILE_OPEN
BUTTON ID_FILE_SAVE
SEPARATOR
BUTTON ID_PEN_THICK_OR_THIN
SEPARATOR
BUTTON ID_FILE_PRINT
BUTTON ID_APP_ABOUT
END
```

利用 ClassWizard 连接命令项识别代码与命令处理函数

新增的三个命令项和一个工具列按钮，都会产生命令消息。接下来的任务就是为它们指定一个对应的命令消息处理例程。下面是一份整理：

```
UI 对象（命令项）         项目识别代码           处理例程
【Pen/Thick Line】  ID_PEN_THICK_OR_THIN  OnPenThickOrThin
【Pen/Pen Widths】  ID_PEN_WIDTHS         OnPenWidths（第10章再处理）
【Edit/Clear All】  ID_EDIT_CLEAR_ALL     OnEditClearAll
```

消息与其处理例程的连接关系是在程序的Message Map中确立，而 Message Map可藉由 ClassWizard 或WizardBar 完成。第 8 章已经利用这两个工具成功地为三个标准的Windows 消息（WM_LBUTTONDOWN、WM_LBUTTONUP、WM_MOUSEMOVE）设立其消息处理函数，现在我们要为 Step2 新增的命令消息设立消息处理例程。过程如下：

- 首先你必须决定，在哪里拦截【Edit/Clear All】才好？本章前面对于消息映射与命令绕行的深度讨论这会儿派上了用场。【Edit/Clear All】这个命令的目的是要清除文件，文件的根本是在数据的「体」，而不在数据的「面」，所以把文件的命令处理例程放在 Document 类中比放在 View 类来得高明。命令消息会不会流经 Document 类？经过前数节的深度之旅，你应该自有定论了。

- 所以，让我们在CScribbleDoc的WizardBar选择【Object IDs】为 ID_EDIT_CLEAR_ALL，并选择【Messages】为COMMAND。

- 猜猜看，如果你在【Object IDs】中选择CScribbleDoc，右侧的【Messages】清单会出现什么？什么都没有！因为Document类只可能接受WM_COMMAND，这一点你应该已经从前面所说的消息递送过程中知道了。如果你在 CScribbleApp的WizardBar上选择【Object IDs】为CScribbleApp，右侧的【Messages】清单中也是什么都没有，道理相同。

- 你会获得一个对话框，询问你是否接受一个新的处理例程。选择Yes，于是文字编辑器中出现该函数之骨干，等待你的幸临...。

这样就完成了命令消息与其处理函数的连接工作。这个工作称为"command binding"。我们的原始代码获得以下修改：

- Document 类之中多了一个函数声明：

```
class CScribbleDoc : public CDocument
{
protected:
    afx_msg void OnEditClearAll();
            ...
}
```

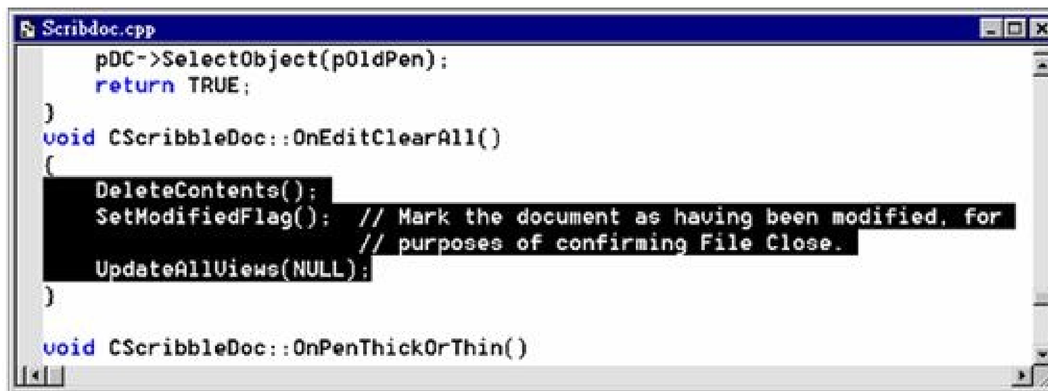
- Document类的Message Map中多了一笔记录：

```
BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
...
END_MESSAGE_MAP()
```

- Document 类中多了一个函数空壳：

```
void CScribbleDoc::OnEditClearAll()
{
}
```

- 现在请写下 OnEditClearAll 函数代码：



依此要领，我们再设计OnPenThickOrThin 函数。此一函数用来更改现行的笔宽，与 Document 有密切关系，所以在Document 类中放置其消息处理例程是适当的：

```
void CScribbleDoc::OnPenThickOrThin()
{
    // Toggle the state of the pen between thin or thick.
    m_bThickPen = !m_bThickPen;
    // Change the current pen to reflect the new user-specified width.
    ReplacePen();
}
void CScribbleDoc::ReplacePen()
{
    m_nPenWidth = m_bThickPen? m_nThickWidth : m_nThinWidth;
    //Change the current pen to reflect the new user-specified width.
    m_penCur.DeleteObject();
    m_penCur.CreatePen(PS_SOLID,m_nPenWidth,RGB(0,0,0));//solid black
}
```

注意，ReplacePen 并非由WizardBar（或ClassWizard）加上去，所以我们必须自行在 CScribbleDoc 类中加上这个函数的声明：

```
class CScribbleDoc : public CDocument
{
protected:
    void ReplacePen();
    ...
}
```

OnPenThickOrThin 函数用来更换笔的宽度，所以CScribbleDoc势必需要加些新的成员变量。变量m_bThickPen 用来记录目前笔的状态（粗笔或细笔），变量m_nThinWidth和 m_nThickWidth 分别记录粗笔和细笔的笔宽——在Step2中此二者固定为2和5，原本并不需要变量的设置，但下一章的 Step3 中粗笔和细笔的笔宽可以更改，所以这里未雨绸缪：

```
class CScribbleDoc : public CDocument
{
// Attributes
protected:
    UINT     m_nPenWidth;     // current user-selected pen width
    BOOL     m_bThickPen;     // TRUE if current pen is thick
    UINT     m_nThinWidth;
    UINT     m_nThickWidth;
    Cpen     m_penCur;        // pen created according to
    ...
}
```

现在重新考虑文件初始化的动作，将Step1的：

```
void CScribbleDoc::InitDocument()
{
    m_nPenWidth = 2; // default 2 pixel pen width
    // solid, black pen
    m_penCur.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0));
}
```

改变为Step2 的：

```
void CScribbleDoc::InitDocument()
{
    m_bThickPen = FALSE;
    m_nThinWidth = 2;   // default thin pen is 2 pixels wide
    m_nThickWidth = 5;  // default thick pen is 5 pixels wide
    ReplacePen();       // initialize pen according to current width
}
```

维护UI对象状态（UPDATE_COMMAND_UI）

上一节我曾提过WizardBar 右侧的【Messages】清单中，针对各个命令项，会出现COMMAND和UPDATE_COMMAND_UI两种选择。后者做什么用？

一个选单拉下来，使用者可以从命令项的状态（打勾或没打勾、灰色或正常）得到一些状态提示。如果Document中没有任何数据的话，【Edit/Clear All】照道理就不应该起作用，因为根本没数据又如何"Clear All" 呢 ?! 这时候我们应该把这个命令项除能（disable）。又例如在粗笔状态下，程序的【Pen/Thick Line】命令项应该打一个勾（所谓的check mark），在细笔状态下不应该打勾。此外，选单命令项的状态应该同步影响到对应之工具列按钮状态。

所有UI对象状态的维护可以依赖所谓的UPDATE_COMMAND_UI 消息。

传统SDK 程序中要改变选单命令项状态，可以调用EnableMenuItem或是CheckMenuItem，但这使得程序杂乱无章，因为你没有一个固定的位置和固定的原则处理命令项状态。MFC 提供一种直觉并且仍旧依赖消息观念的方式，解决这个问题，这就是UPDATE_COMMAND_UI 消息。其设计理念是，每当选单被拉下并尚未显示之前，其命令项（以及对应之工具列按

钮）都会收到 UPDATE_COMMAND_UI 消息，这个消息和 WM_COMMAND 有一样的绕行路线，我们（程序员）只要在适当的类中放置其处理函数，并在函数中做某些判断，便可决定如何显示命令项。

这种方法的最大好处是，不但把问题的解决方式统一化，更因为 Framework传给UPDATE_COMMAND_UI处理例程的参数是一个「指向CCmdUI对象的指针」，而CCmdUI对象就代表着对应的选单命令项，因此你只需调用CCmdUI 所准备的，专门用来处理命令项外观的函数（如 Enable 或 SetCheck）即可。我们的工作量大为减轻。

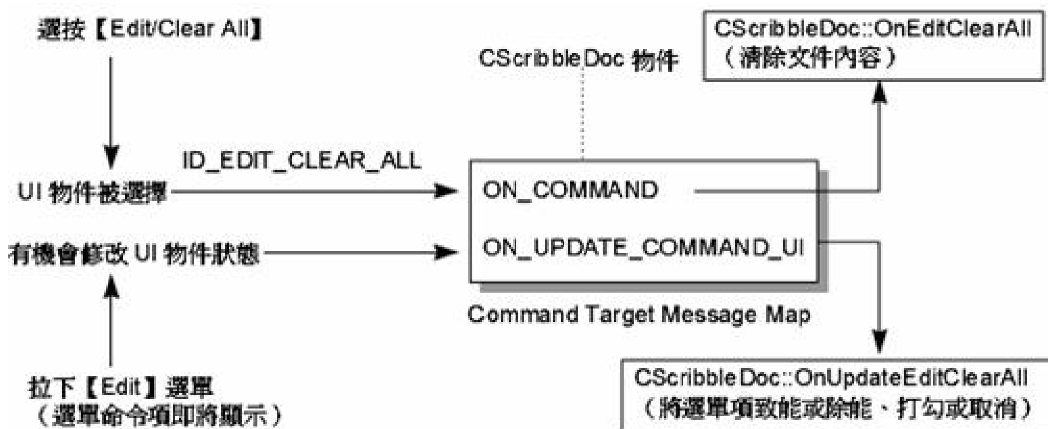


图9-7 ON_COMMAND和ON_UPDATE_COMMAND_UI的运作

图9-7以【Edit/Clear All】实例说明ON_COMMAND和 ON_UPDATE_COMMAND_UI的运作。为了拦截UPDATE_COMMAND_UI 消息，你的 Command Target 对象（也许是Application，也许是windows，也许是Views，也许是Documents）要做两件事情：

1. 利用WizardBar（或ClassWizard）加上一笔Message Map 项目如下：

ON_UPDATE_COMMAND_UI(ID_xxx, OnUpdatexxx)

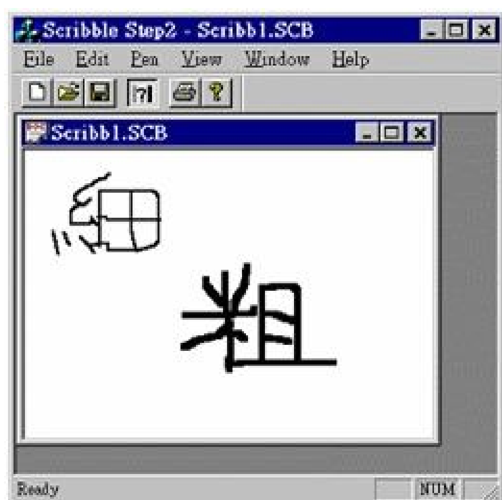
2. 提供一个OnUpdatexxx 函数。这个函数的写法十分简单，因为 Framework 传来一个代表UI对象（也就是选单命令项或工具列按钮）的 CCmdUI 对象指针，而对UI对象的各种操作又都已设计在CCmdUI成员函数中。举个例子：

```
void CScribbleDoc::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!m_strokeList.IsEmpty());
}
void CScribbleDoc::OnUpdatePenThickOrThin(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_bThickPen);
}
```

如果命令项与某个工具列按钮共享同一个命令ID，上述的Enable动作将不只影响命令项，也影响按钮。命令项的打勾（checked）即是按钮的按下（depressed），命令项没有打勾（unchecked）即是按钮的正常化（松开）。

现在，Scribble 第二版全部修改完毕，制作并测试之：

- 在整合环境中按下【Build/Build Scribble】编译并链接。
- 按下【Build/Execute】执行 Scribble。测试细笔粗笔的运作情况，以及【Edit /Clear All】是否生效。



从写程序（而不是挖背后意义）的角度去看 Message Map，我把Step2 所进行的选单改变对 Message Map 造成的影响做个总整理。一共有四个相关成份会被ClassWizard（或 WizardBar）产生出来，下面就是相关原始代码，其中只有第 4 项的函数内容是我们撰写的，其它都由工具自动完成。

1. CSRIBBLEDOC.CPP

```
BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
//{{AFX_MSG_MAP(CScribbleDoc)
ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
ON_COMMAND(ID_PEN_THICK_OR_THIN, OnPenThickOrThin)
ON_UPDATE_COMMAND_UI(ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll)
ON_UPDATE_COMMAND_UI(ID_PEN_THICK_OR_THIN,OnUpdatePenThickOrThin)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

不要去掉//{{ 和//}}，否则下次ClassWizard或WizardBar不能正常工作。

2. CSRIBBLEDOC.H

```
class CScribbleDoc : public CDocument
{
    ...
    // Generated message map functions
    protected:
    //{{AFX_MSG(CScribbleDoc)
    afx_msg void OnEditClearAll();
    afx_msg void OnPenThickOrThin();
    afx_msg void OnUpdateEditClearAll(CCmdUI* pCmdUI);
    afx_msg void OnUpdatePenThickOrThin(CCmdUI* pCmdUI);
    //}}AFX_MSG
    ...
};
```

3. RESOURCE.H

```
#define ID_PEN_THICK_OR_THIN 32772
#define ID_PEN_WIDTHS 32773
```

（另一个项目ID_EDIT_CLEAR_ALL已经在AFXRES.H中定义了）

4. SCRIBBLEDOC.CPP

```
void CScribbleDoc::OnEditClearAll()
{
    DeleteContents();
    SetModifiedFlag(); //Mark the document as having been modified,for
    //purposes of confirming File Close.
    UpdateAllViews(NULL);
}
void CScribbleDoc::OnPenThickOrThin()
{
    // Toggle the state of the pen between thin or thick.
    m_bThickPen = !m_bThickPen;
    // Change the current pen to reflect the new user-specified width.
    ReplacePen();
}
void CScribbleDoc::ReplacePen()
{
    m_nPenWidth = m_bThickPen? m_nThickWidth : m_nThinWidth;
    // Change the current pen to reflect the new user-specified width.
    m_penCur.DeleteObject();
    m_penCur.CreatePen(PS_SOLID,m_nPenWidth,RGB(0,0,0));//solid black
}
void CScribbleDoc::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
    // Enable the command user interface object (menu item or tool bar
    // button) if the document is non-empty, i.e., has at least one stroke.
    pCmdUI->Enable(!m_strokeList.IsEmpty());
}
void CScribbleDoc::OnUpdatePenThickOrThin(CCmdUI* pCmdUI)
{
    // Add check mark to Draw Thick Line menu item, if the current
    // pen width is "thick".
    pCmdUI->SetCheck(m_bThickPen);
}
```

本章回顾

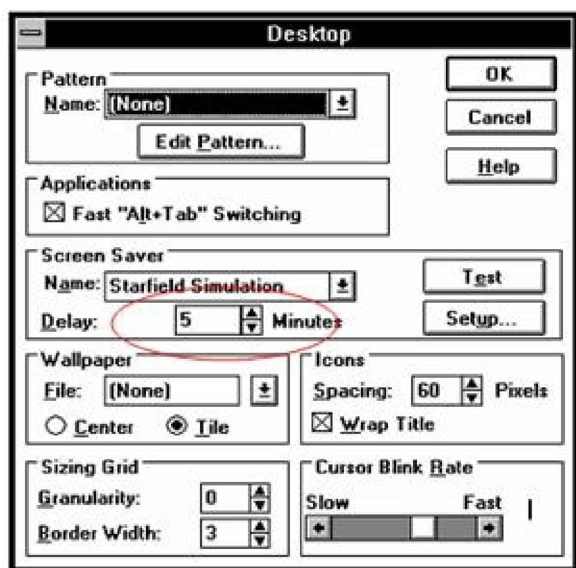
这一章主要为Scribble Step2 增加新的选单命令项。在这个过程中我们使用了工具列编辑器和ClassWizard（或Wizardbar）等工具。工具的使用很简单，但是把消息的处理常式加在什么地方却是关键。因此本章一开始先带你深入探索MFC原始代码，了解消息的递送以及所谓Message Map背后的意义，并且也解释了命令消息（WM_COMMAND) 特异的绕行路线及其原因。

我在本章中挖出了许多MFC原始代码，希望藉由原始代码的自我说明能力，加深你对消息映射与消息绕行路径的了解。这是对 MFC「知其所以然」的重要关键。这个知识基础不会因为MFC的原始代码更动而更动，我要强调的，是其原理。

第10章 MFC与对话框

上一章我们为 Scribble 新增了一个【Pen】选单，其中第二个命令项【Pen Width...】准备用来提供一个对话框，让使用者设定笔的宽度。每一线条都可以拥有自己的笔宽。原预设粗笔是 5 个图素宽，细笔是 2 个图素宽。

为了这样的目的，在对话框中放个 Spin 控制组件是极佳的选择。Spin 就是那种有着上下小三角形箭头、可搭配一个文字显示器的控制组件，有点像转轮，用来选择数字最合适：



但是，Scribble Step3 只是想示范如何在MFC程序中经由选单命令项唤起一个对话框，并示范所谓的数据交换与数据检验（DDX/DDV）。所以，笔宽对话框中只选用两个小小的 Edit 控制组件而已。

本章还可以学习到如何利用对话框编辑器设计对话框的面板，并利用 ClassWizard 制作一个对话框类，定义消息处理函数，把它们与对话框「绑」在一块儿。

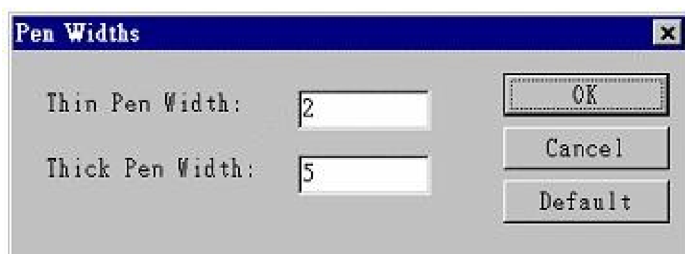


图10-1 【Pen Widths】对话框

对话框编辑器

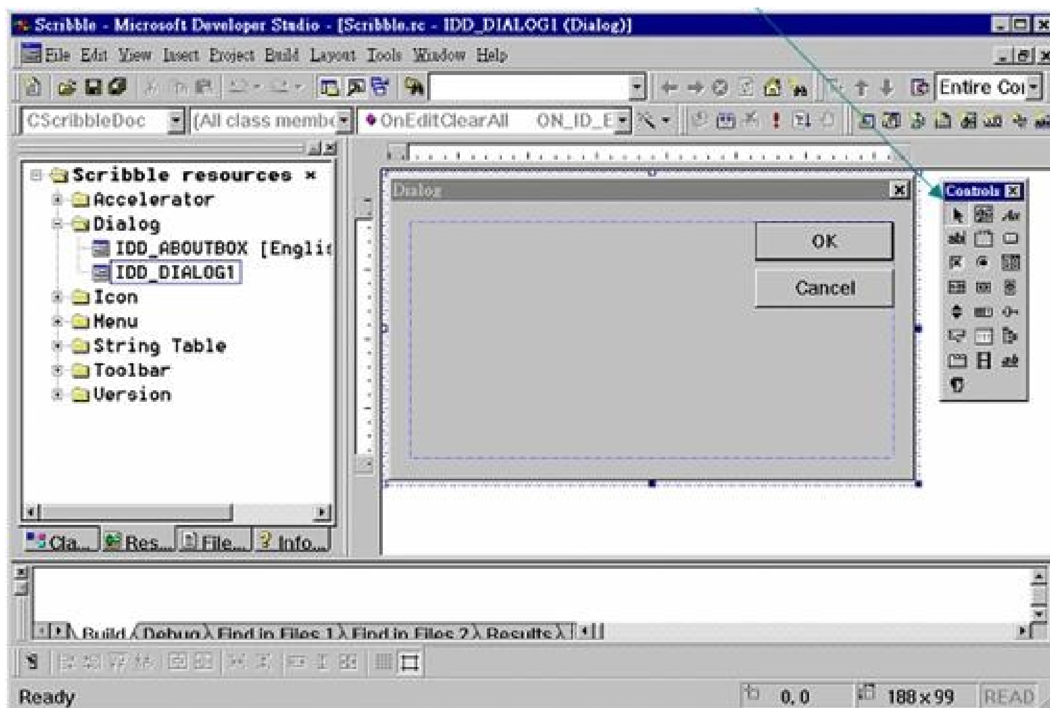
把对话框函数抛在一旁，把所有程序烦恼抛在一旁，我们先享受一下Visual C++ 整合环境中的对话框编辑器带来的对话框面板（Dialog Template）设计快感。

设计对话框面板，有两个重要的步骤，第一是从工具箱中选择控制组件（control，功能各异的小小零组件）加到对话框中，第二是填写此一控制组件的标题、ID、以及其它性质。

以下就是利用对话框编辑器设计【Pen Widths】对话框的过程。

+在Visual C++整合环境中选按【Insert/Resource】命令项，并在随后而来的【Insert Resource】对话框中，选择【resource types】为Dialog。

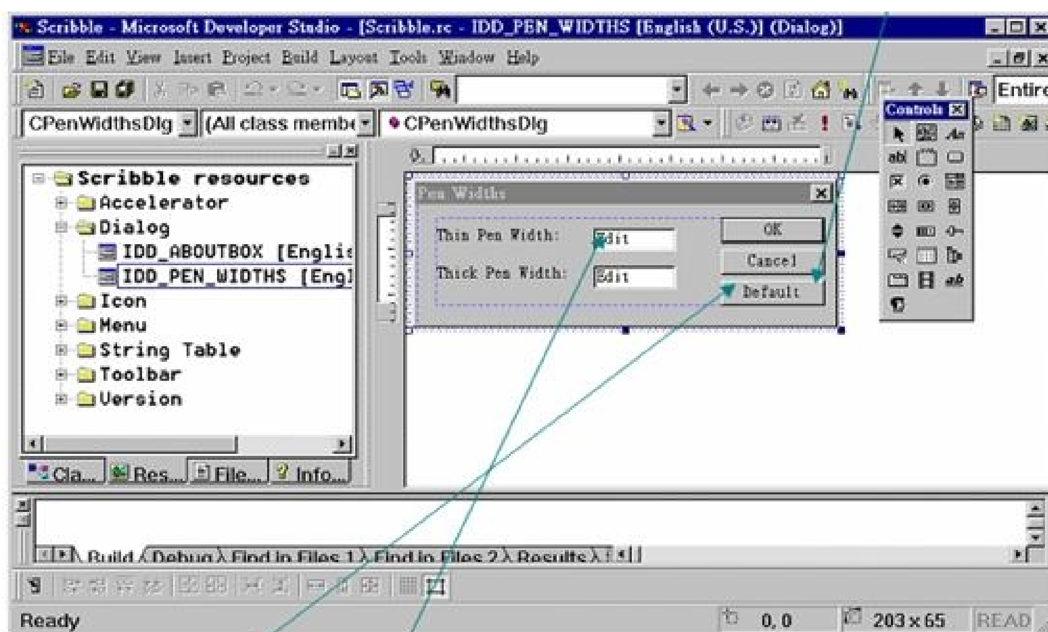
- 或是直接在Visual C++整合环境中按下工具列的【New Dialog】按钮。
- Scribble.rc 文件会被打开，对话框编辑器出现，自动给我们一个空白对话框，内含两个按钮，分别是【OK】和【Cancel】。控制组件工具箱出现在画面右侧，内含许多控制组件。



- 为了设定控制组件的属性，必须用到【Dialog Properties】对话框。如果它最初没有出现，只要以右键选按对话框的任何地方，就会跑出一份选单，再选择其中的Properties，即会出现此对话框。按下对话框左上方的push-pin 钮（大头针）可以常保它浮现为最上层窗口。现在把对话框ID改为IDD_PEN_WIDTHS，把标题改为"Pen Widths"。



- 为对话框加入两个Edit控制组件，两个Static 控制组件，以及一个按钮。



- 右键选按新增的按钮，在Property page中把其标题改为"Default"，并把ID改为IDC_DEFAULT_PEN_WIDTHS。

- 右键选按第一个Edit控制元件，在Property page中把ID改为IDC_THIN_PEN_WIDTH。以同样的方式把第二个 Edit 控制组件的 ID 改为 IDC_THICK_PEN_WIDTH。



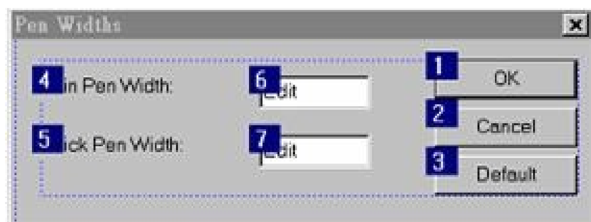
- 右键选按第一个Static控制组件，Property page 中出现其属性，现在把文字内容改为"Thin Pen Width: "。以同样的方式把第二个Static 控制组件的文字内容改为"Thick Pen Width:"。不必在意Static控制组件的ID值，因为我们根本不可能在程序中用到Static控制组件的ID。



- 调整每一个控制组件的大小位置，使之美观整齐。

- 调整tab order。所谓tab order是使用者在操作对话框时，按下Tab键后，键盘输入焦点在各个控制组件上的巡回次序。调整方式是选按Visual C++ 整合环境中的【Layout/Tab Order】命令项，出现带有标号的对话框如下，再依你所想要的次序以鼠标点选一遍即

可。



- 测试对话框。选按Visual C++ 整合环境中的【Layout/Test】命令项，出现运作状态下的对话框。你可以在这种状态下测试tab order和预设按钮（ default button）。若欲退出，请选按【OK】或【Cancel】或按下ESC键。

注意：所谓default button，是指与 `<Enter>` 键相通的那个按钮。

所有调整都完成之后，存盘。于是SCRIBBLE.RC 增加了下列内容（一个对话框面板）：

```
IDD_PEN_WIDTHS DIALOG DISCARDABLE  0, 0, 203, 65
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Pen Widths"
FONT 8, "MS Sans Serif"
BEGIN
DEFPUSHBUTTON   "OK",IDOK,148,7,50,14
PUSHBUTTON      "Cancel",IDCANCEL,148,24,50,14
PUSHBUTTON      "Default",IDC_DEFAULT_PEN_WIDTHS,148,41,50,14
LTEXT           "Thin Pen Width:",IDC_STATIC,10,12,70,10
LTEXT           "Thick Pen Width:",IDC_STATIC,10,33,70,10
EDITTEXT        IDC_THIN_PEN_WIDTH,86,12,40,13,ES_AUTOHSCROLL
EDITTEXT        IDC_THICK_PEN_WIDTH,86,33,40,13,ES_AUTOHSCROLL
END
```

利用ClassWizard连接对话框与其专属类

一旦完成了对话框的外貌设计，再来就是设计其行为。我们有两件事要做：

1. 从 MFC 的 CDialog 中派生出一个类，用来负责对话框行为。

2. 利用 ClassWizard 把这个类和先前你产生的对话框资源连接起来。通常这意味着你必须声明某些函数，用以处理你感兴趣的对话框消息，并将对话框中的控制组件对应到类的成员变量上，这也就是所谓的Dialog Data eXchange（DDX）。如果你对这些变量内容有任何「确认规则」的话，ClassWizard 也允许你设定之，这就是所谓的 Dialog Data Validation（DDV)。

注意：所谓「确认规则」是指对某些特殊用途的变量进行内容查验工作。例如月份一定只可能在1~12 之间，日期一定只可能在1~31 之间，人名一定不会有数字夹杂其中，金钱数额不能夹带文字，新竹的电话号代码必须是03开头后面再加 7 位数...等等等。

所有动作当然都可以手工完成，然而ClassWizard 的表现非常好，让我们快速又轻松地完成这些事样。它可以为你的对话框产生一个 .H 檔，一个 .CPP 文件，内有你的对话框类、函数骨干、一个Message Map、以及一个Data Map。哎呀，我们又看到了新东西，稍后我会解释所

谓的 Data Map。

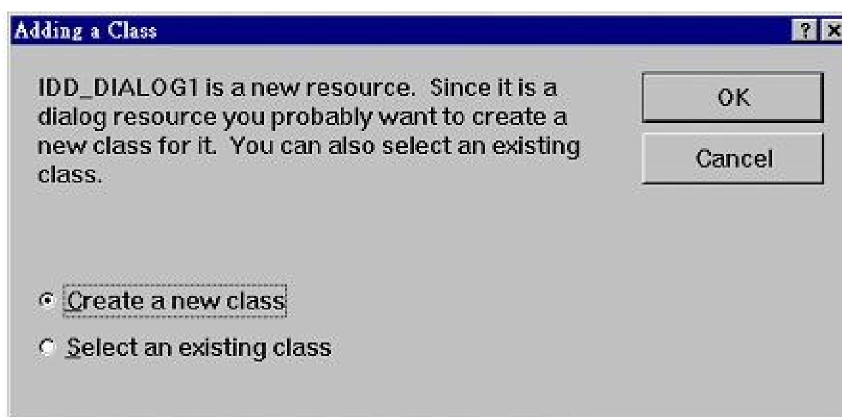
回忆Scribble诞生之初，程序中有一个About对话框，寄生于 SCRIBBLE.CPP 中。AppWizard并没有询问我们有关这个对话框的任何意见，就自作主张地放了这些代码：

```
#0001 ////////////////////////////////////////////////////////////////////////
#0002 // CAboutDlg dialog used for App About
#0003
#0004 class CAboutDlg : public CDialog
#0005 {
#0006 public:
#0007         CAboutDlg();
#0008
#0009 // Dialog Data
#0010         //{{AFX_DATA(CAboutDlg)
#0011         enum { IDD = IDD_ABOUTBOX };
#0012         //}}AFX_DATA
#0013
#0014         // ClassWizard generated virtual function overrides
#0015         //{{AFX_VIRTUAL(CAboutDlg)
#0016         protected:
#0017         virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
#0018         //}}AFX_VIRTUAL
#0019
#0020 // Implementation
#0021 protected:
#0022         //{{AFX_MSG(CAboutDlg)
#0023                 // No message handlers
#0024         //}}AFX_MSG
#0025         DECLARE_MESSAGE_MAP()
#0026 };
#0027
#0028 CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
#0029 {
#0030         //{{AFX_DATA_INIT(CAboutDlg)
#0031         //}}AFX_DATA_INIT
#0032 }
#0033
#0034 void CAboutDlg::DoDataExchange(CDataExchange* pDX)
```

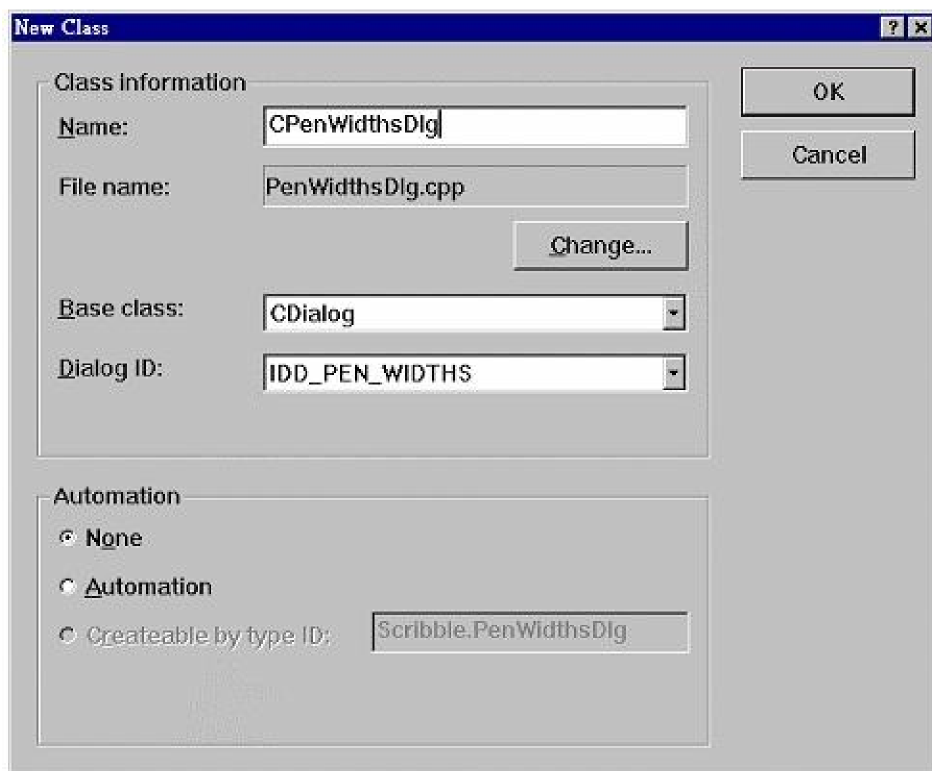
```
#0035 {
#0036         CDialog::DoDataExchange(pDX);
#0037         //{{AFX_DATA_MAP(CAboutDlg)
#0038         //}}AFX_DATA_MAP
#0039 }
#0040
#0041 BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
#0042         //{{AFX_MSG_MAP(CAboutDlg)
#0043                 // No message handlers
#0044         //}}AFX_MSG_MAP
#0045 END_MESSAGE_MAP()
#0046
#0047 // App command to run the dialog
#0048 void CScribbleApp::OnAppAbout()
#0049 {
#0050         CAboutDlg aboutDlg;
#0051         aboutDlg.DoModal();
#0052 }
```

CAboutDlg虽然派生自CDialog，但太简陋，不符合我们新增的这个【Pen Width】对话框所需，所以我们首先必须另为【Pen Width】对话框产生一个类，以负责其行径。步骤如下：

- 接续刚才完成对话框面板的动作，选按整合环境的【View/ClassWizard】命令项（或是直接在对话框面板上快按两下），进入 ClassWizard。这时候【Adding a Class】对话框会出现，并以刚才的 IDD_PEN_WIDTHS 为新资源，这是因为ClassWizard 知道你已在对话框编辑器中设计了一个对话框面板，却还未设计其对应类（整合环境就是这么便利）。好，按下【OK】。



- 在【Create New Class】对话框中设计新类。键入 "CPenWidthsDlg" 做为类名称。请注意类的基础类型为CDialog，因为ClassWizard 知道目前是由对话框编辑器过来：



- ClassWizard 把类名称再加上 .cpp 和 .h，作为预设文件名。毫无问题，因为Windows 95和Windows NT 都支持长文件名。如果你不喜欢，按下上图右侧的【Change】钮去改它。本例改用PENDLG.CPP和PENDLG.H 两个文件名。
- 按下上图的【OK】钮，于是类产生，回到ClassWizard 画面。

这样，我们就进账了两个新文件：

PENDLG.H

```
#0001  // PenDlg.h : header file
#0002  //
#0003
#0004  /////////////////////////////////////////////////////////////////////////////
#0005  // CPenWidthsDlg dialog
#0006
#0007  class CPenWidthsDlg : public CDialog
#0008  {
#0009  // Construction
#0010  public:
#0011          CPenWidthsDlg(CWnd* pParent = NULL);   // standard constructor
#0012
#0013  // Dialog Data
#0014          //{{AFX_DATA(CPenWidthsDlg)
#0015          enum { IDD = IDD_PEN_WIDTHS };
#0016                  // NOTE: the ClassWizard will add data members here
#0017          //}}AFX_DATA
#0018
#0019
#0020  // Overrides
#0021          // ClassWizard generated virtual function overrides
#0022          //{{AFX_VIRTUAL(CPenWidthsDlg)
#0023          protected:
#0024          virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV suppo
#0025          //}}AFX_VIRTUAL
#0026
#0027  // Implementation
#0028  protected:
#0029
#0030          // Generated message map functions
#0031          //{{AFX_MSG(CPenWidthsDlg)
#0032          afx_msg void OnDefaultPenWidths();
#0033          //}}AFX_MSG
#0034          DECLARE_MESSAGE_MAP()
#0035  };
```

PENDLG.CPP

```
#0001  // PenDlg.cpp : implementation file
#0002  //
#0003
#0004  #include "stdafx.h"
#0005  #include "Scribble.h"
```

```
#0006 #include "PenDlg.h"
#0007
#0008 #ifdef _DEBUG
#0009 #define new DEBUG_NEW
#0010 #undef THIS_FILE
#0011 static char THIS_FILE[] = __FILE__;
#0012 #endif
#0013
#0014 /////////////////////////////////////////////////////////////////////
#0015 // CPenWidthsDlg dialog
#0016
#0017
#0018 CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
#0019        : CDialog(CPenWidthsDlg::IDD, pParent)
#0020 {
#0021        //{{AFX_DATA_INIT(CPenWidthsDlg)
#0022           // NOTE: the ClassWizard will add member initialization here
#0023        //}}AFX_DATA_INIT
#0024 }
#0025
#0026
#0027 void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
#0028 {
#0029        CDialog::DoDataExchange(pDX);
#0030        //{{AFX_DATA_MAP(CPenWidthsDlg)
#0031               // NOTE: the ClassWizard will add DDX and DDV calls here
#0032        //}}AFX_DATA_MAP
#0033 }
#0034
#0035
#0036 BEGIN_MESSAGE_MAP(CPenWidthsDlg, CDialog)
#0037        //{{AFX_MSG_MAP(CPenWidthsDlg)
#0038        ON_BN_CLICKED(IDC_DEFAULT_PEN_WIDTHS, OnDefaultPenWidths)
#0039        //}}AFX_MSG_MAP
#0040 END_MESSAGE_MAP()
#0041
#0042 /////////////////////////////////////////////////////////////////////
#0043 // CPenWidthsDlg message handlers
#0044
#0045 void CPenWidthsDlg::OnDefaultPenWidths()
#0046 {
#0047        // TODO: Add your control notification handler code here
#0048
#0049 }
```

稍早我曾提过，ClassWizard 会为我们做出一个 Data Map。此一Data Map将放在DoDataExchange 函数中。目前Data Map还没有什么内容，CPenWidthsDlg 的 Message Map也是空的，因为我们还未透过ClassWizard 加料呢。

请注意，CPenWidthsDlg 构造函数会先引发基类 CDialog 的构造函数，后者会产生一个modal对话框。CDialog 构造函数的两个参数分别是对话框ID以及父窗口指针：

```
#0018  CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
#0019        : CDialog(CPenWidthsDlg::IDD, pParent)
#0020  {
#0021      //{{AFX_DATA_INIT(CPenWidthsDlg)
#0022      // NOTE: the ClassWizard will add member initialization here
#0023      //}}AFX_DATA_INIT
#0024  }
```

ClassWizard帮我们把CPenWidthsDlg::IDD塞给第一个参数，这个值 定 义 于PENDLG.H 的 AFX_DATA 区中，其值为 IDD_PEN_WIDTHS：

```
#0013  // Dialog Data
#0014      //{{AFX_DATA(CPenWidthsDlg)
#0015      enum { IDD = IDD_PEN_WIDTHS };
#0016          // NOTE: the ClassWizard will add data members here
#0017      //}}AFX_DATA
```

也就是【Pen Widths】对话框资源的 ID：

```
// in SCRIBBLE.RC
IDD_PEN_WIDTHS DIALOG DISCARDABLE  0, 0, 203, 65
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Pen Widths"
FONT 8, "MS Sans Serif"
BEGIN
DEFPUSHBUTTON   "OK",IDOK,148,7,50,14
PUSHBUTTON      "Cancel",IDCANCEL,148,24,50,14
PUSHBUTTON      "Default",IDC_DEFAULT_PEN_WIDTHS,148,41,50,14
LTEXT           "Thin Pen Width:",IDC_STATIC,10,12,70,10
LTEXT           "Thick Pen Width:",IDC_STATIC,10,33,70,10
EDITTEXT        IDC_THIN_PEN_WIDTH,86,12,40,13,ES_AUTOHSCROLL
EDITTEXT        IDC_THICK_PEN_WIDTH,86,33,40,13,ES_AUTOHSCROLL
END
```

对话框类CPenWidthsDlg 因此才有办法取得「RC 档中的对话框资源」。

对话框的消息处理函数

CDialog本就定义有两个按钮【OK】和【Cancel】，【Pen Widths】对话框又新增一个【Default】钮，当使用者按下此钮时，粗笔与细笔都必须回复为预设宽度（分别是 5 个图素和 2 个图素）。那么，我们显然有两件工作要完成：

1. 在CPenWidthsDlg 中增加两个变量，分别代表粗笔与细笔的宽度。

2. 在CPenWidthsDlg 中增加一个函数，负 责【Default】钮被按下后的动作。

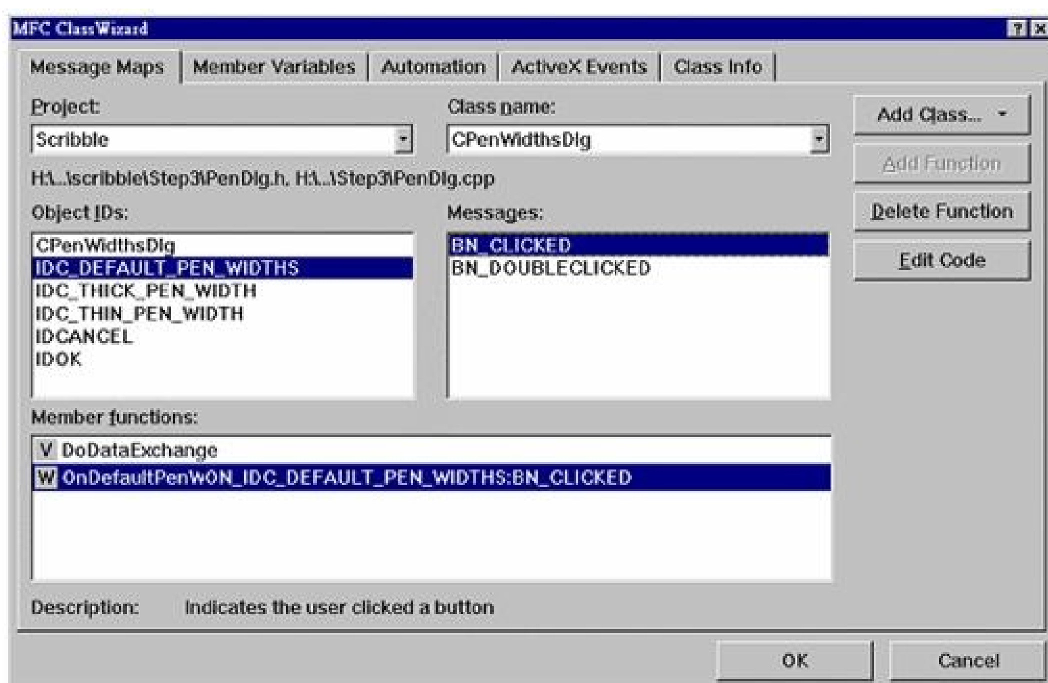
以下是ClassWizard 的操作步骤（增加一个函数）：

- 进入ClassWizard，选择【Message Maps】附页，再选择【Class name】清单中的 CPenWidthsDlg。

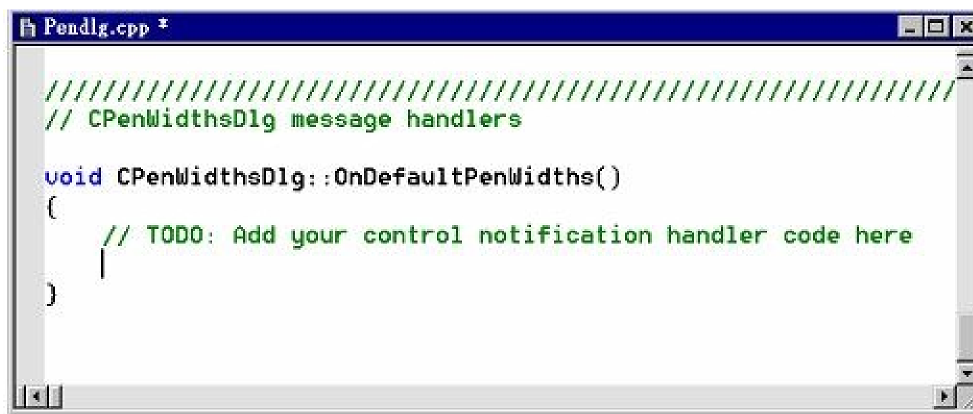
- 左侧的【 Object IDs】清单列出对话框中各个控制组件的 ID。请选择其中的 IDC_DEFAULT_PEN_WIDTHS（代表【Default】钮）。
- 在右侧的【Messages】中选择 BN_CLICKED。这和我们在前两章的经验不同，如今我们处理的是控制组件，它所产生的消息是特别的一类，称为 Notification消息，这种消息是控制组件用来通知其父窗口（通常是个对话框）某些状况发生了，例如BN_CLICKED 表示按钮被按下。至于不同的 Notification 所代表的意义，画面最下方的"Description" 会显示出来。
- 按下【Add Function】钮，接受预设的 OnDefaultPenWidths 函数（也可以改名）：



- 现在，【Member Functions】清单中出现了新函数，以及它所对映之控制组件与 Notification 消息。



- 按下【Edit Code】钮，光标落在OnDefaultPenWidths 函数身上，我们看到以下内容：



上述动作对原始代码造成的影响是：

```
// in PENDLG.H
class CPenWidthsDlg : public CDialog
{
protected:
afx_msg void OnDefaultPenWidths();
...
};
// in PENDLG.CPP
BEGIN_MESSAGE_MAP(CPenWidthsDlg, CDialog)
ON_BN_CLICKED(IDC_DEFAULT_PEN_WIDTHS, OnDefaultPenWidths)
END_MESSAGE_MAP()
void CPenWidthsDlg::OnDefaultPenWidths()
{
    // TODO : Add your control notification handler here
}
```

MFC 中各式各样的MAP

如果你以为MFC中只有Message Map和Data Map，那你就错了。另外还有一个Dispatch Map，使用于OLE Automation，下面是其形式：

```
DECLARE_DISPATCH_MAP()  // .H 文件中的宏，声明 Dispatch Map。
BEGIN_DISPATCH_MAP(CClikDoc, CDocument) // .CPP 檔中的 Dispatch Map
//{{AFX_DISPATCH_MAP(CClikDoc)
DISP_PROPERTY(CClikDoc, "text", m_str, VT_BSTR)
DISP_PROPERTY_EX(CClikDoc, "x", GetX, SetX, VT_I2)
DISP_PROPERTY_EX(CClikDoc, "y", GetY, SetY, VT_I2)
//}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

此外还有Event Map，使用于OLE Custom Control（也就是OCX），下面是其形式：

```
DECLARE_EVENT_MAP()  // .H 文件中的宏，声明 Event Map。
BEGIN_EVENT_MAP(CSmileCtrl, COleControl) // .CPP 檔中的 Event Map
//{{AFX_EVENT_MAP(CSmileCtrl)
EVENT_CUSTOM("Inside", FireInside, VTS_I2  VTS_I2)
EVENT_STOCK_CLICK()
//}}AFX_EVENT_MAP
END_EVENT_MAP()
```

至于Message Map，我想你一定已经很熟悉了：

```
DECLARE_MESSAGE_MAP()// .H 文件中的宏，声明Message Map。
BEGIN_MESSAGE_MAP(CScribDoc, CDocument) // .CPP 檔中的 Message Map
//{{AFX_MSG_MAP(CScribDoc)
ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
ON_COMMAND(ID_PEN_THICK_OR_THIN, OnPenThickOrThin)
ON_COMMAND(ID_PEN_WIDTHS, OnPenWidths)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

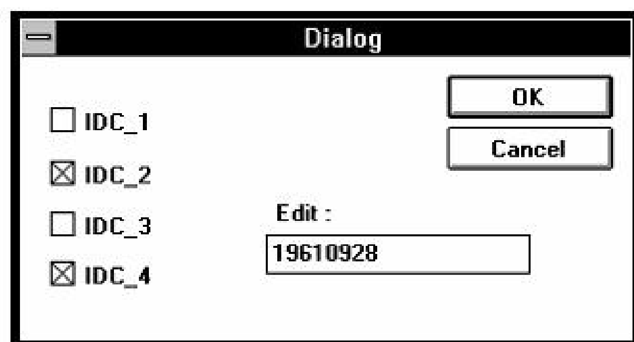
MFC所谓的Map，其实就是一种类似表格的东西，它的背后是什么？可能是一个巨大的数据结构（例如Message Map）。最和其它Map形式不同的，就属 Data Map了，它的形式是：

```
//{{AFX_DATA_MAP(CPenWidthsDlg)  // .CPP 檔中的 Data Map
DDX_Text(pDX, IDC_THIN_PEN_WIDTH, m_nThinWidth);
DDV_MinMaxInt(pDX, m_nThinWidth, 1, 20);
DDX_Text(pDX, IDC_THICK_PEN_WIDTH, m_nThickWidth);
DDV_MinMaxInt(pDX, m_nThickWidth, 1, 20);
//}}AFX_DATA_MAP
```

针对同一个数据目标（成员变量），Data Map 之中每组有两笔记录，一笔负责DDX，一笔负责DDV。

对话框数据交换与查核（DDX & DDV）

在解释 DDX/DDV 的来龙去脉之前，我想先描述一下 SDK 程序处理对话框数据的作法。如果你设计一个对话框如下图：



当【OK】钮被按下，程序应该一一取得按钮状态以及Edit内容：

```
char _OpenName[128];
GetDlgItemText(hwndDlg, IDC_EDIT, _OpenName, 128);
If (IsDlgButtonChecked(hDlg,IDC_1))
    ...;
If (IsDlgButtonChecked(hDlg,IDC_2))
    ...;
If (IsDlgButtonChecked(hDlg,IDC_3))
    ...;
If (IsDlgButtonChecked(hDlg,IDC_4))
    ...;
// hDlg 代表对话框的窗口 handle
```

虽然Windows 95和Windows NT有所谓的通用型对话框（Common Dialog，第 6 章末尾曾介绍过），某些个标准对话框的设计因而非常简单，但非标准的对话框还是得像上面那样自己动手。

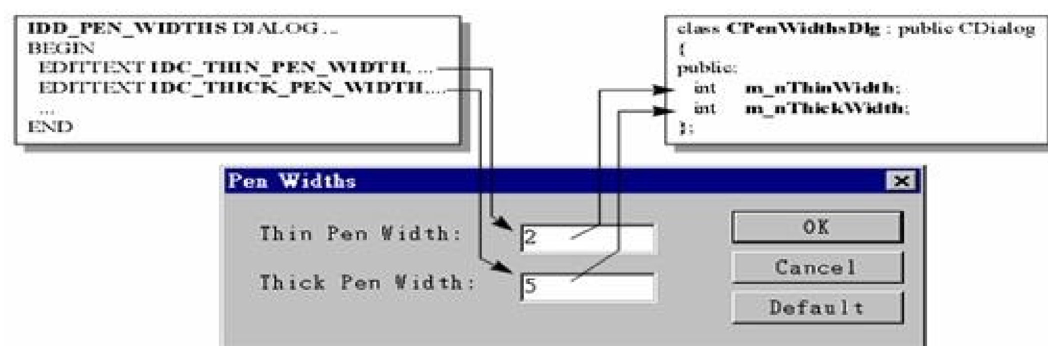
MFC的方式就简单多了。它提供的DDX（X 表示 eXchange），允许程序员事先设定控制组件与变量的对应关系。我们不但可以令控制组件的内容一有改变就自动传送到变量去，也可以藉MFC提供的DDV（V 表示Validation）设定字段的合理范围。如果使用者在字段上键入超出合理范围的数字，就会在按下【OK】后出现类似以下的画面：



数据的查核（Data Validation）其实是一件琐碎又耗人力的事情，各式各样的数据都应该要检查其合理范围，程序才算面面俱到。例如日期字段绝不能允许12 以上的月份以及31 以上的日子（如果程序还能自动检查 2 月份只有 28 天而遇闰年有 29 天那就更棒了）；金额字段里绝不能允许文字出现，电话号代码字段一定只有 9 位（至少台湾目前是如此）。为了解决这些琐碎又累人的工作，市售有一些链接库，专门做数据查核工作。

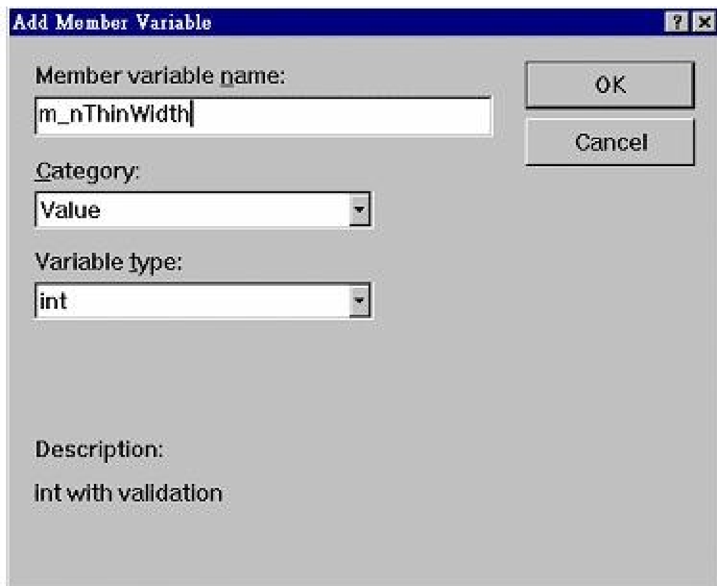
然而不要对 MFC 的 DDV 能力期望过高，稍后你就会看到，它只能满足最低层次的要求而已。就 DDV 而言，Borland 的OWL表现较佳。

现在我打算以两个成员变量映射到对话框上的两个Edit字段。我希望当使用者按下【OK】钮，第一个Edit字段的内容自动储存到m_nThinWidth变量中，第二个Edit栏位的内容自动储存到m_nThickWidth 变量中：

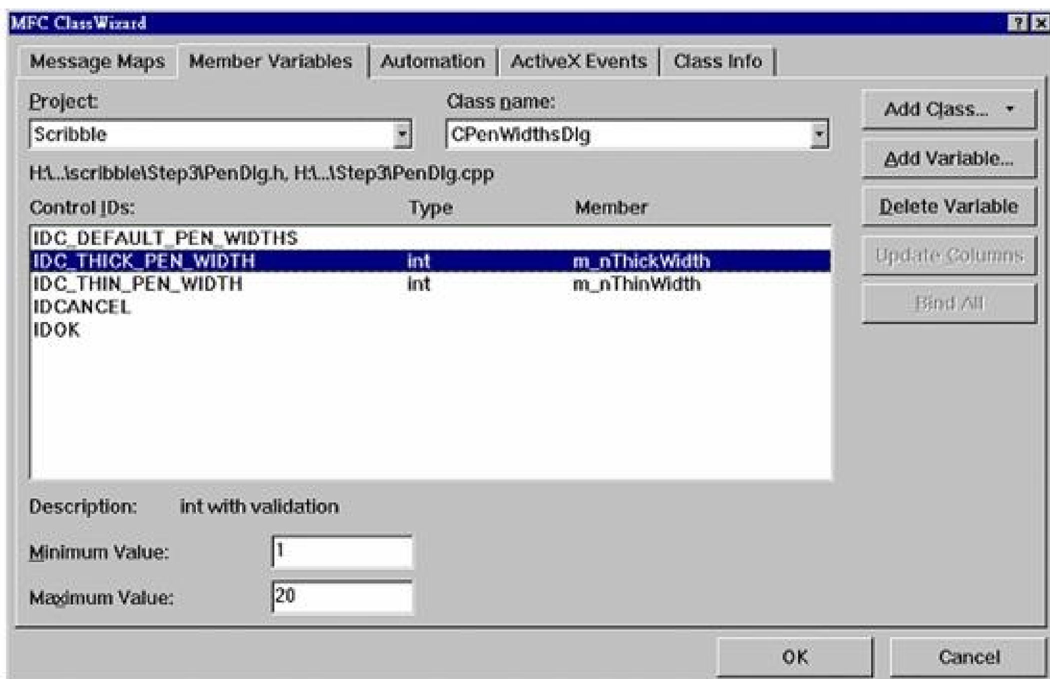


下面是 ClassWizard 的操作步骤（为对话框类增加两个成员变量，并设定 DDX / DDV）：

- 进入ClassWizard，选择【Member Variables】附页，再选择 CPenWidthsDlg。对话框中央部分有一大块局部用来显示控制组件与变量间的对映关系（见下一页图）。
- 选择IDC_THIN_PEN_WIDTH，按下【Add Variable...】钮，出现对话框如下。
- 键入变量名称为m_nThinWidth。
- 选择变量型别为int 。



- 按下【OK】键， 于是ClassWizard为CPenWidthsDlg 增加了一个变量m_nThinWidth。
- 在ClassWizard 对话框最下方（见下一页图）填入变量的数值范围，以为 DDV之用。
- 重复前述步骤，为IDC_THICK_PEN_WIDTH 也设定一个对应变量，范围也是1~20。



上述动作影响我们的程序代码如下：

```
class CPenWidthsDlg : public CDialog
{
// Dialog Data
//{{AFX_DATA(CPenWidthsDlg)
enum { IDD = IDD_PEN_WIDTHS };
int m_nThinWidth;
int m_nThickWidth;
//}}AFX_DATA
...
CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
: CDialog(CPenWidthsDlg::IDD, pParent)
{
    m_nThickWidth = 0;
    m_nThinWidth = 0;
    ...
}
void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CPenWidthsDlg)
    DDX_Text(pDX, IDC_THIN_PEN_WIDTH, m_nThinWidth);
    DDV_MinMaxInt(pDX, m_nThinWidth, 1, 20);
    DDX_Text(pDX, IDC_THICK_PEN_WIDTH, m_nThickWidth);
    DDV_MinMaxInt(pDX, m_nThickWidth, 1, 20);
    //}}AFX_DATA_MAP
}
```

只要数据「有必要」在成员变量与控制组件之间搬移，Framework 就会自动调用DoDataExchange。我所说的「有必要」是指，对话框初次显示在屏幕上，或是使用者按下【OK】离开对话框等等。CPenWidthsDlg::DoDataExchange 由一组一组的DDX/DDV函数完成之。先做DDX，然后做DDV，这是游戏规则。如果你纯粹借助ClassWizard，就不必在意此事，如果你要自己动手完成，就得遵循规则。

该是完成上一节的OnDefaultPenWidths 的时候了。当【Default】钮被按下，Framework会调用OnDefaultPenWidths，我们应该在此设定粗笔细笔两种宽度的默认值：

```
void CPenWidthsDlg::OnDefaultPenWidths()
{
    m_nThinWidth = 2;
    m_nThickWidth = 5;
    UpdateData(FALSE);  // causes DoDataExchange()
    // bSave=FALSE means don't save from screen,
    // rather, write to screen
}
```

MFC 中各式各样的DDx_函数

如果你以为MFC对于对话框的照顾，只有DDX和DDV，那你就又错了，另外还有一个DDP，使用于OLE Custom Control（也就是OCX）的Property page 中，下面是它的形式：

```
//{{AFX_DATA_MAP(CSmilePropPage)
DDP_Text(pDX, IDC_CAPTION, m_caption, _T("Caption") );
DDX_Text(pDX, IDC_CAPTION, m_caption);
DDP_Check(pDX, IDC_SAD, m_sad, _T("sad") );
DDX_Check(pDX, IDC_SAD, m_sad);
//}}AFX_DATA_MAP
```

什么是Property page？这是最新流行（Microsoft 强力推销？）的界面。这种界面用来解决过于拥挤的对话框。ClassWizard 就有四个 Property page，我们又称为tag（附页）。拥有 property page 的对话框称为 property sheet，也就是 tagged dialog（带有附页的对话框）。

如何唤起对话框

【Pen Widths】对话框是一个所谓的 Modal 对话框，意思是除非它关闭（结束），否则它会紧抓住这个程序的控制权，但不影响其它程序。相对于Modal对话框，有一种Modeless 对话框就不会影响程序其它动作的进行；通常你在文字处理软件中看到的文字搜寻对话框就是 Modeless 对话框。

过去， MFC有两个类，分别负责Modal对话框和Modeless对话框，它们是CModalDialog和CDialog。如今已经合并为一，就是CDialog。不过为了回溯相容，MFC 有这么一个定义：

```
#define CModalDialog Cdialog
```

要做出Modal对话框，只要调用CDialog::DoMoal 即可。

我们希望Step3 的命令项【Pen/Pen Widths】被按下时，【Pen Widths】对话框能够执行起来。要唤起此一对话框，得做到两件事情：

1. 产生一个 CPenWidthsDlg 对象，负责管理对话框。

2. 显示对话框窗口。这很简单，调用DoMoal 即可办到。

为了把命令消息连接上CPenWidthsDlg，我们再次使用ClassWizard，这一次要为 CScribbleDoc 加上一个命令处理例程。为什么选择在 CScribbleDoc 而不是其它类中处理此一命令呢？因为不论是粗笔或细笔，乃至于目前正使用的笔，其宽度都被记录在 CScribbleDoc 中成为它的一个成员变量：

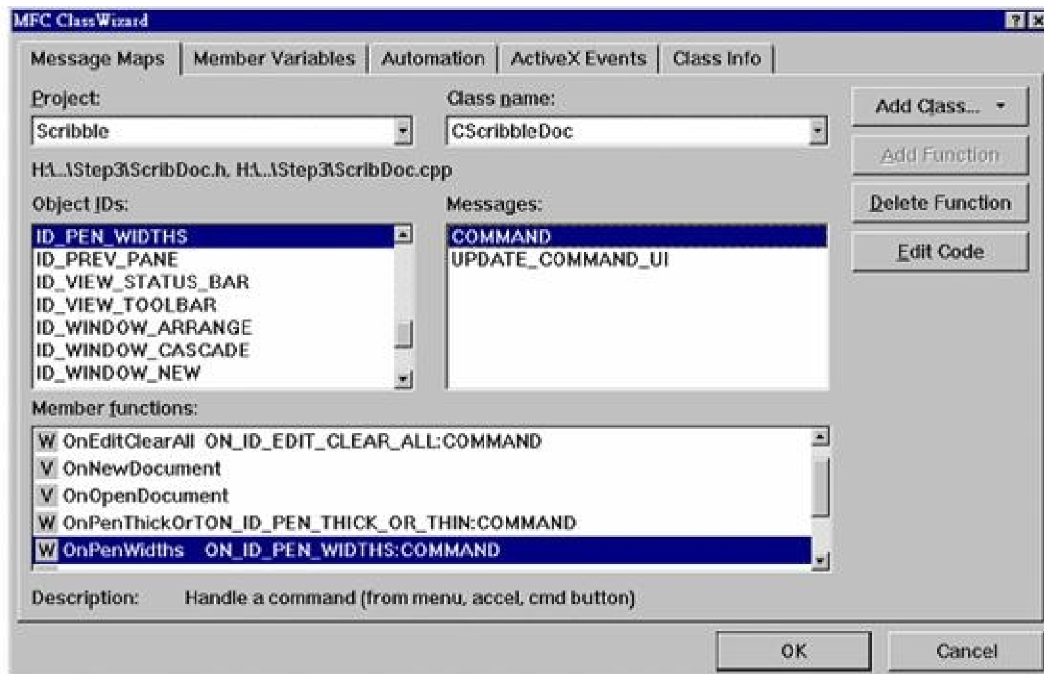
```
// in SCRIBDOC.H
class CScribbleDoc : public CDocument
{
protected:
    UINT    m_nPenWidth;  //current user-selected pen width
    UINT    m_nThinWidth;
    UINT    m_nThickWidth;
    ...
}
```

所以由CScribDoc负责唤起对话框，接受笔宽设定，是很合情合理的事。

如果命令消息处理例程名为OnPenWidths，我们希望在这个函数中先唤起对话框，由对话框取得粗笔和细笔的宽度，然后再把这两个值设定给 CScribbleDoc 中的两个对应变量。下面是设计步骤。

- 执行ClassWizard，选择【Message Map】附页，并选择CScribbleDoc。

- 在【Object IDs】清单中选择ID_PEN_WIDTHS。
- 在【Messages】清单中选择COMMAND。
- 按下【Add Function】钮并接受ClassWizard 给予的函数名称 OnPenWidths。



- 按下【Edit Code】钮，游标落在 OnPenWidths 函数内，键入以下内容：

```
// SCRIBDOC.CPP
#include "pendlg.h"
...
void CScribbleDoc::OnPenWidths()
{
    CPenWidthsDlg dlg;
    // Initialize dialog data
    dlg.m_nThinWidth = m_nThinWidth;
    dlg.m_nThickWidth = m_nThickWidth;
    // Invoke the dialog box
    if (dlg.DoModal() == IDOK)
    {
        // retrieve the dialog data
        m_nThinWidth = dlg.m_nThinWidth;
        m_nThickWidth = dlg.m_nThickWidth;
        //Update the pen that is used by views when drawing new strokes,
        //to reflect the new pen width definitions for "thick" and "thin".
        ReplacePen();
    }
}
```

现在，Scribble Step3 全部完成，制作并测试之。



本章回顾

上一章我们为Scribble加上三个新的选单命令项。其中一个命令项【Pen/Pen Widths...】将引发对话框，这个目标在本章实现。

制作对话框，我们需要为此对话框设计面板（Dialog Template），这可藉 Visual C++整合环境之对话框编辑器之助完成。我们还需要一个派生自 CDialog 的类（本例为 CPenWidthsDlg）。ClassWizard 可以帮助我们新增类，并增加该类的成员变量，以及设定对话框之 DDX/DDV 。以上都是透过 ClassWizard 以鼠标点点选选而完成，过程中不需要写任何一进程序代码。

所谓DDX是让我们把对话框类中的成员变量与对话框中的控制组件产生关联，于是当对话框结束时，控制组件的内容会自动传输到这些成员变量上。

所谓DDV是允许我们设定对话框控制组件的内容类型以及数据（数值) 范围。

对话框的写作，在MFC程序设计中轻松无比。你可以尝试练习一个比较复杂的对话框。

第11章 View功能之加强与重绘效率之提升

前面数章中，我们已经看到了View 如何扮演Document与使用者之间的媒介：View 显示 Document 的数据内容，并且接受鼠标在窗口上的行为（左键按下、放开、鼠标移动），视为对 Document 的操作。

Scribble可以对同一份Document产生一个以上的Views，这是MDI程序的「天赋」 MDI 程序标准的【Window/New Window】窗体项目就是为达此目标而设计的。但有一个缺点还待克服，那就是你在窗口 A 的绘图动作不能实时影响窗口 B，也就是说它们之间并没有所谓的同步更新——即使它们是同一份数据的一体两面！

Scribble Step4 解决上述问题。主要关键在于想办法通知所有相同血源（同一份Document）的各兄弟（各个Views），让它们一起行动。但却因此必须考虑这个问题：

如果使用者的一个鼠标动作引发许多许多的程序绘图动作，那么「同步更新」的绘图效率就变得非常重要。因此在考虑如何加强显示能力时，我们就得设计所谓的「必要绘图区」，也就是所谓的Invalidate Region，或称「不再适用的局部」或「重绘区」。每当使用者增加新的线条，Scribble Step4 便把「包围该线条之最小四方形」设定为「必要绘图区」。为了记录这项数据，从 Step1 延用至今的 Document 数据结构必须有所改变。

Step4 的同步更新，是以一笔画为单位，而非以一个点为单位。换句话说在一笔画未完成之前，不打算让同源的多个 View 窗口同步更新 -- 那毕竟太伤效率了。

Scribble Step4 的另一项改善是为 Document Frame 窗口增加垂直和水平滚动条，并且示范一种所谓的分裂窗口（Splitter window），如图 11-1。这种窗口的主要功能是当使用者欲对文件做一体两面（或多面）观察时，各个观察子窗口可以集中在一个大的母窗口中。在这里，子窗口被称为「窗口」（pane）。

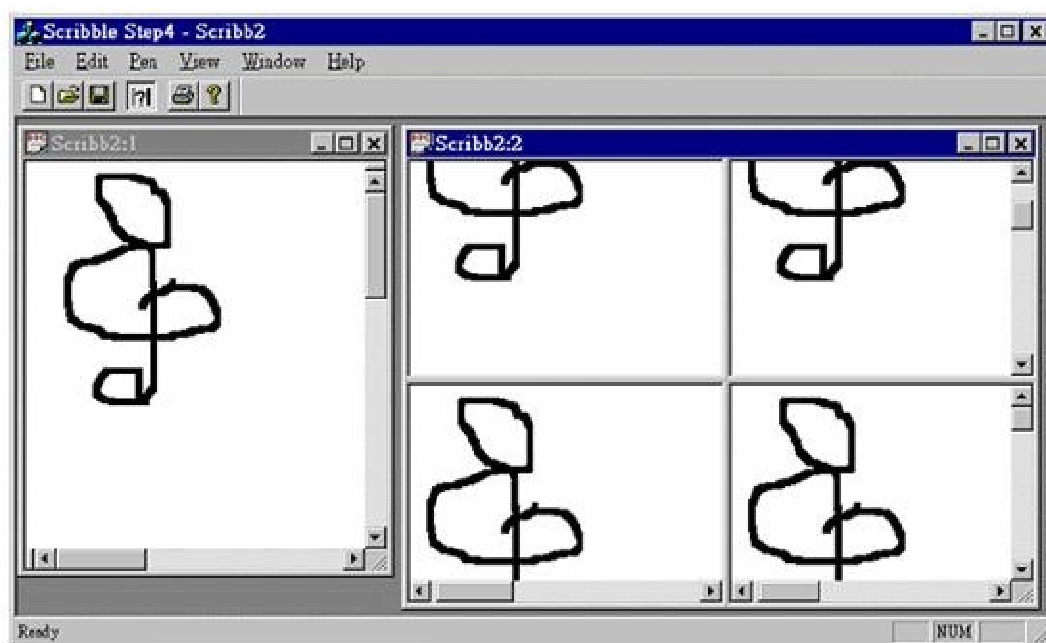


图11-1 Scribble Step4，同源（同一份Document）之各个View 窗口具备同步更新的能力。Document Frame窗口具备分裂窗口与卷轴。

同时修改多个 Views：UpdateAllViews 和 OnUpdate

在Scribble View上绘图，然后选按【Window/New Window】，会蹦出另一个新的View，其内的图形与前一个View 相同。这两个Views 就是同一份文件的两个「观景窗」。新窗口的产生导至 WM_PAINT 产生，于是 OnDraw 发生效用，把文件内容画出来：

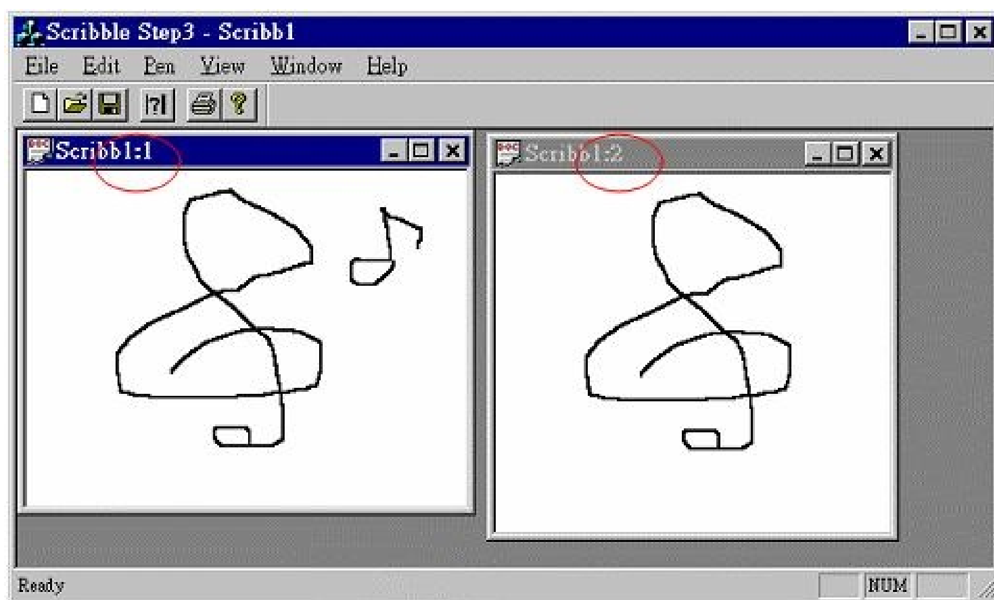


图11-2 一份Document连结两个Views，没有同步修正画面

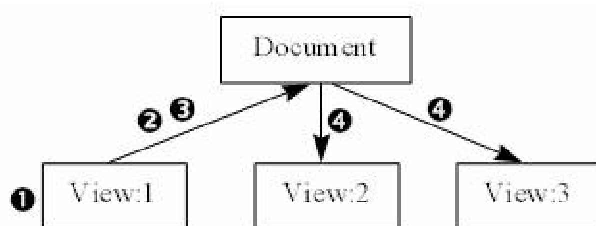
但是，此后如果你在Scrib1:1窗口上绘图而未缩放其尺寸的话（也就是不产生WM_PAINT），Scrib1:2 窗口内看不到后续绘图内容。我们并不希望如此，不幸的是上一章的Scribble Step3正是如此。

不能同步更新的关键在于，没有人通知所有的兄弟们（Views）一起动手——动手调用OnDraw。你是知道的，只有当WM_PAINT产生，OnDraw才会被调用。因此，解决方式是对每一个兄弟都发出WM_PAINT，或任何其它方法——只要能通知到就好。也就是说，让附属于同一Document的所有Views都能够立即反应 Document内容变化的方法就是，始作俑者（被使用者用来修改Document内容的那个View）必须想办法通知其他兄弟。

经由CDocument::UpdateAllViews，MFC提供了这样的一个标准机制。让所有的Views 同步更新数据的关键在于两个函数：

1.CDocument::UpdateAllViews——这个函数会巡访所有隶属同一份 Document 的各个Views，找到一个就通知一个，而所谓「通知」就是调用其 OnUpdate 函数。

2.CView::OnUpdate ——我们可以在这个函数中设计绘图动作。或许是全部重绘（这比较笨一点），或许想办法只绘必要的一小部分（这比较聪明一些）。



- 1 使用者在View:1 做动作（View 扮演使用者接口的第一线）。
- 2 View:1调用GetDocument，取得Document指针，更改数据内容。
- 3 View:1调用Document 的UpdateAllViews。
- 4 View:2和View:3 的OnUpdate一一被调用起来，这是更新画面的时机。

如果想让绘图程序聪明一些，不要每次都全部重绘，而是只择「必须重绘」的局部重绘，那么OnUpdate 需要被提示什么是「必须重绘的局部」，这就必须借助于 UpdateAllViews的参数：

```
virtual void UpdateAllViews(CView* pSender,
LPARAM lHint,
CObject* pHint);
```

- 第一个参数代表发出此一通牒的始作俑者。这个参数的设计无非是希望避免重复而无谓的通牒，因为始作俑者自己已经把画面更新过了（在鼠标消息处理常式中），不需要再被通知。
- 后面两个参数lHint和pHint是所谓的提示参数（Hint），它们会被传送到同一Document所对应的每一个Views的OnUpdate函数去。lHint可以是一些特殊的提示值，pHint则是一个派生自CObject的对象指针。靠着设计良好的「提示」，OnUpdate 才有机会提高绘图效率。要不然直接通知OnDraw就好了，也不需要再搞出一个OnUpdate。

另一方面，OnUpdate 收到三个参数（由 CDocument:: UpdateAllViews 发出）：

```
virtual void OnUpdate(CView* pSender,
LPARAM lHint,
CObject* pHint);
```

因此，一旦Document 数据改变，我们应该调用 CDocument::UpdateAllVi

ews以通知所有相关的Views。而在CMyView::OnUpdate函数中我们应该以效率为第一考虑，利用参数中的hint设定重绘区，使后续被唤起的OnDraw有最快的工作速度。注意，通常你不应该在OnUpdate中执行绘图动作，所有的绘图动作最好都应该集中在OnDraw；你在OnUpdate 函数中的行为应该是计算哪一块区 域需要重绘，然后调用CWnd::InvalidateRect，发出WM_PAINT让OnDraw去画图。结论是，改善同步更新以及绘图效率的前置工作如下：

1. 定义hint 的数据类型，用以描述已遭修改的数据局部。

2. 当使用者透过View改变了Document 内容，程序应该产生一个hint，描述此一修改，并以它做为参数，调用UpdateAllViews。

3. 改写CMyView::OnUpdate，利用hint设计高效率绘图动作，使hint 描述区之外的局部不要重画。

在 View 中定义一个 hint

以Scribble 为例，当使用者加上一段线条，如果我们计算出包围此一线条之最小四方形，那么只有与此四方形有交集的其它线条才需要重画，如图 11-3。因此在Step4中把hint设计为RECT类型，差堪为用。

效率考虑上，当然我们还可以精益求精取得各线条与此四方形的交点，然后只重绘四方形内部的那一部分即可，但这么做是否动用太多计算，是否使工程太过复杂以至于不划算，你可得谨慎评估。

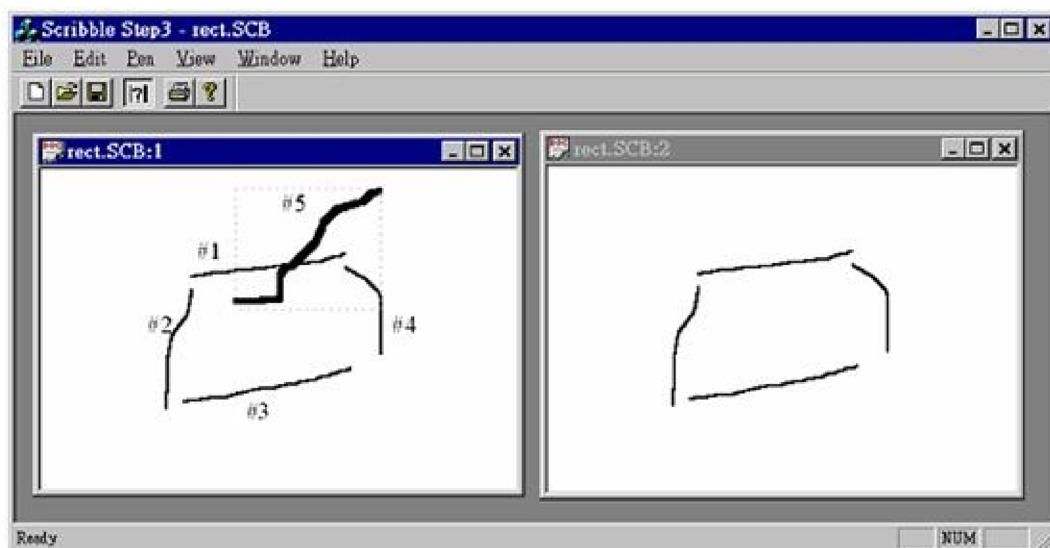


图11-3 在rect.SCB:1窗口中新增一线条#5，那么，只有与虚线四方形（此四方形将#5 包起来）有交集之其它线条，也就是#1和#4，才有必要在rectSCB:2 窗口中重画。

前面曾说UpdateAllViews 函数的第三个参数必须是CObject派生对象之指针。由于本例十分单纯，与其为了Hint特别再设计一个类，勿宁在CStroke中增加一个变量（事实上是一个 CRect 对象），用以表示前述之hint四方形，那么每一条线条就外罩了一个小小的四方壳。但是我们不能把CRect对象指针直接当做参数来传，因为CRect并不派生自 CObject。稍后我会说明该怎么做。

可以预期的是，日后一定需要一一从每一线条中取出这个「外围四方形」，所以现在先声明并定义一个名为GetBoundingRect 的函数。另外再声明一个 FinishStroke 函数，其作用主要是计算这四方形尺寸。稍后我会详细解释这些函数的行为。

```
// in SCRIBBLEDOC.H
class CStroke : public CObject
{
    ...
public:
    UINT         m_nPenWidth;
    CDWordArray  m_pointArray;
    CRect        m_rectBounding; //smallest rect that surrounds all
    //of the points in the stroke
public:
    CRect& GetBoundingRect() { return m_rectBounding; }
    void FinishStroke();
    ...
};
```

我想你早已熟悉了Scribble Document 的数据结构。为了应付Step4的需要，现在每一线条必须多一个成员变量，那是一个CRect对象，如图11-4所示。

```
CScribble Step4 Document：
```

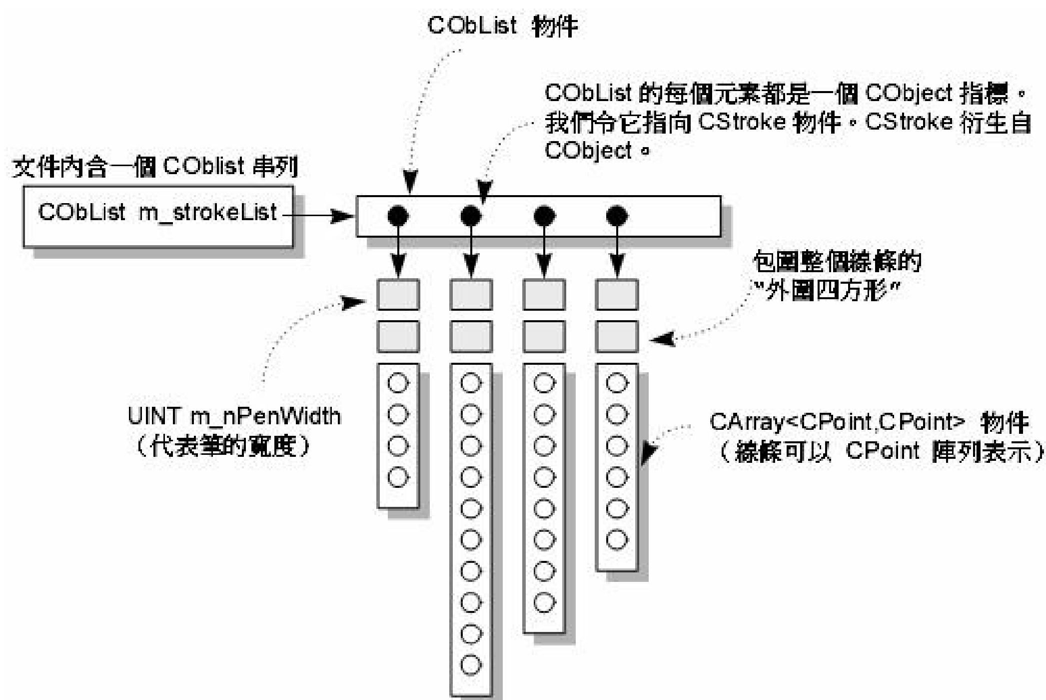


图11-4 CScribbleDoc对象中的各项数据

设计观念分析完毕，现在动手吧。我们必须在SCRIBDOC.CPP中的Document 初始化动作以及文件读写动作都加入m_rectBounding 这个新成员：

```
// in SCRIBDOC.CPP
IMPLEMENT_SERIAL(CStroke, CObject, 2)  // 注意schema no.改变为2
CStroke::CStroke(UINT nPenWidth)
{
    m_nPenWidth = nPenWidth;
    m_rectBounding.SetRectEmpty();
}
void CStroke::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_rectBounding;
        ar << (WORD)m_nPenWidth;
        m_pointArray.Serialize(ar);
    }
    else
    {
        ar >> m_rectBounding;
        WORD w;
        ar >> w;
        m_nPenWidth = w;
        m_pointArray.Serialize(ar);
    }
}
```

如果我们改变了文件读写的格式，我们就应该改变schema number（可视为版本号代码）。由于Scribble 数据文件（.SCB）格式改变了，多了一个 m_rectBounding，所以我们在 IMPLEMENT_SERIAL 宏中改变Document的 Schema no.， 以便让不同版本的Scribble 程序识得不同版本的文件文件。如果你以Scribble Step3读取Step4所产生的文件，会因为Schema 号代码的不同而得到这样的消息：



我们还需要一个函数，用以计算「线条之最小外包四方形」，这件事情当然是在线条完成后进行之，所以此一函数命名为FinishStroke。每当一笔画结束（鼠标左键放开，产生 WM_LBUTTONUP），OnLButtonUp 就调用FinishStroke让它计算边界。计算方法很直接，取出线条中的坐标点，比大小而已：

```
// in SCRIBDOC.CPP
void CStroke::FinishStroke()
{
    // 计算外围四方形。为了灵巧而高效率的重绘动作，这是必要的。
    if (m_pointArray.GetSize()==0)
    {
        m_rectBounding.SetRectEmpty();
        return;
    }
    CPoint pt = m_pointArray[0];
    m_rectBounding = CRect(pt.x, pt.y, pt.x, pt.y);
    for (int i=1; i < m_pointArray.GetSize(); i++)
    {
        // 如果点在四方形之外，那么就将四方形膨胀，以含入该点。
        pt = m_pointArray[i];
        m_rectBounding.left = min(m_rectBounding.left, pt.x);
        m_rectBounding.right = max(m_rectBounding.right, pt.x);
        m_rectBounding.top = min(m_rectBounding.top, pt.y);
        m_rectBounding.bottom = max(m_rectBounding.bottom, pt.y);
    }
    // 在四方形之外再加上笔的宽度。
    m_rectBounding.InflateRect(CSize(m_nPenWidth, m_nPenWidth));
    return;
}
```

把 hint传给OnUpdate

下一步骤是想办法把hint交给UpdateAllViews。让我们想想什么时候 Scribble的数据开始产生改变？答案是鼠标左键按下时！或许你会以为要在 OnLButtonDown中调用CDocument::UpdateAllViews。这个猜测的论点可以成立但是结果错误，因为左键按下后还有一连串的鼠标轨迹移动，每次移动都导至数据改变，新的点不断被加上去。如果我们等左键放开，线条完成，再来调用 UpdateAllViews，事情会比较单纯。因此Scribble Step4是在OnButtonUp 中调用UpdateAllViews。当然我们现在就可以预想得到，一笔画完成之前，同一Document 的其它Views 没有办法实时显示最新数据。

下面是 OnButtonUp 的修改内容：

```
void CScribbleView::OnLButtonUp(UINT, CPoint point)
{
    ...
    m_pStrokeCur->m_pointArray.Add(point);
    // 已完成加点的动作，现在可以计算外围四方形了
    m_pStrokeCur->FinishStroke();
    // 通知其它的 views，使它们得以修改它们的图形。
    pDoc->UpdateAllViews(this, 0L, m_pStrokeCur);
    ...
    return;
}
```

程序逻辑至为简单，唯一需要说明的是UpdateAllViews的第三个参数（hint），原本我们只需设此参数为m_rectBounding，即可满足需求，但MFC规定，第三参数必须是一个CObject 指针，而CRect并不派生自CObject，所以我们干脆就把目前正作用中的整个线条（也就是m_pStrokeCur）传过去算了。CStroke 的确是派生自CObject！

```
// in SCRIBBLEVIEW.H
class CScribbleView : public CScrollView
{
protected:
    CStroke*   m_pStrokeCur; // the stroke in progress
    ...
};
// in SCRIBBLEVIEW.CPP
void CScribbleView::OnLButtonDown(UINT, CPoint point)
{
    ...
    m_pStrokeCur = GetDocument()->NewStroke();
    m_pStrokeCur->m_pointArray.Add(point);
    ...
}
void CScribbleView::OnMouseMove(UINT, CPoint point)
{
    ...
    m_pStrokeCur->m_pointArray.Add(point);
    ...
}
void CScribbleView::OnLButtonUp(UINT, CPoint point)
{
    ...
    m_pStrokeCur->m_pointArray.Add(point);
    m_pStrokeCur->FinishStroke();
    pDoc->UpdateAllViews(this, 0L, m_pStrokeCur);
    ...
}
```

UpdateAllViews会巡访CScribbleDoc 所连接的每一个Views（始作俑者那个View 除外），调用它们的OnUpdate函数，并把hint做为参数之一传递过去。

利用 hint 增加重绘效率

预设情况下，OnUpdate 所收到的无效区（也就是重绘区），是 Document Frame 窗口的整个内部。但谁都知道，原已存在而且没有变化的图形再重绘也只是浪费时间而已。上一节你已看到Scribble每加上一整个线条，就在 OnLButtonUp函数中调用UpdateAllViews 函数，并且把整个线条（内含其四方边界）传过去，因此我们可以想办法在 OnUpdate 中重绘这个四方形小局部就好。

话说回来，如何能够只重绘一个小局部就好呢？我们可以一一取出 Document中每一线条的四方边界，与新线条的四方边界比较，若有交点就重绘该线条。CRect 有一个IntersectRect 函数正适合用来计算四方形交集。

但是有一点必须注意，绘图动作不是集中在OnDraw 吗？因此OnUpdate和 OnDraw之间的分工有必要厘清。前面数个版本中这两个函数的动作是：

- OnUpdate——啥也没做。事实上CScribbleView 原本根本没有改写此一函数。
- OnDraw——迭代取得Document中的每一线条，并调用 CStroke::DrawStroke 将线条绘出。

Scribble Step4 之中，这两个函数的动作如下：

- OnUpdate——判断Framework传来的hint是否为CStroke 对象。如果是，设定无效局部（重绘局部）为该线条的外围四方形；如果不是，设定无效局部为整个窗口局部。「设定无效局部」（也就是调用 CWnd::InvalidateRect）会引发WM_PAINT，于是引发OnDraw。
- OnDraw——迭代取得Document中的每一线条，并调用CStroke::GetBound

ingRect取线条之外围四方形，如果与「无效局部」有交集，就调用 CStroke::DrawStroke绘出整个线条。如果没有交集，就跳过不画。

以下是新增的 OnUpdate 函数：

```
// in SCRIBVW.CPP
void CScribbleView::OnUpdate(CView* /* pSender */, LPARAM /* lHint */,
CObject* pHint)
{
    // Document 通知 View 说，某些数据已经改变了
    if (pHint != NULL)
    {
        if (pHint->IsKindOf(RUNTIME_CLASS(CStroke)))
        {
            //hint提示我们哪一线条被加入（或被修改），所以我们要把该线条的
            // 外围四方形设为无效区。
            CStroke* pStroke = (CStroke*)pHint;
            CClientDC dc(this);
            OnPrepareDC(&dc);
            CRect rectInvalid = pStroke->GetBoundingRect();
            dc.LPtoDP(&rectInvalid);
            InvalidateRect(&rectInvalid);
            return;
        }
    }
    // 如果我们不能解释 hint 内容（也就是说它不是我们所预期的
    // CStroke 对象），那就让整个窗口重绘吧（把整个窗口设为无效区）。
    Invalidate(TRUE);
    return;
}
```

为什么OnUpdate 之中要调用OnPrepareDC？这关系到滚动条，我将在介绍分裂窗口时再说明。另，GetBoundingRect 动作如下：

```
CRect& GetBoundingRect() { return m_rectBounding; }
```

OnDraw函数也为了高效能重绘动作之故，做了以下修改。阴影部分是与 Scribble Step3不同之处：

```
// SCRIBVW.CPP
void CScribbleView::OnDraw(CDC* pDC)
{
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // 取得窗口的无效区。如果是在打印状态情况下，则取
    // printer DC 的截割区（clipping region）。
    CRect rectClip;
    CRect rectStroke;
    pDC->GetClipBox(&rectClip);
    // 注意：CScrollView::OnPrepare 已经在 OnDraw 被调用之前先一步
    // 调整了 DC 原点，用以反应出目前的卷动位置。关于 CScrollView，
    // 下一节就会提到。
    // 调用 CStroke::DrawStroke 完成无效区中各线条的绘图动作
    CTypedPtrList<CObList,CStroke*>& strokeList = pDoc->m_strokeList;
    POSITION pos = strokeList.GetHeadPosition();
    while (pos != NULL)
    {
        CStroke* pStroke = strokeList.GetNext(pos);
        rectStroke = pStroke->GetBoundingRect();
        if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
        continue;
        pStroke->DrawStroke(pDC);
    }
}
```

可卷动的窗口：CScrollView

到目前为止我们还没有办法观察一张比窗口还大的图，因为我们没有滚动条。一个View窗口没有滚动条，是很糟糕的事，因为通常Document 范围大而观景窗范围小。我们不能老让Document与View 窗口一样大。一个具备滚动条的View 窗口更具有「观景窗」的意义。

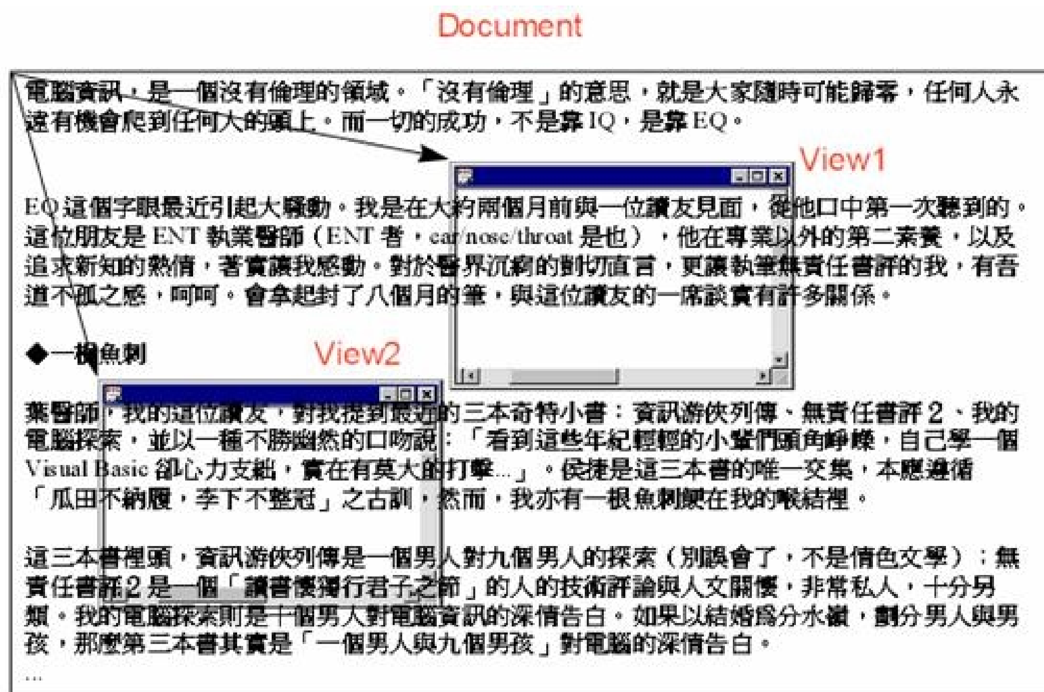


图11-5a 一个具备滚动条的View窗口更具「观景窗」的意义

如果你有SDK程序设计经验，你就会知道设计一个可卷动的窗口是多么烦琐的事（文字的卷动还算好，图形的卷动更惨）。MFC 当然不可能对此一般性功能坐视不管，事实上它已设计好一个CScrollView，其中的滚动条有实时卷动（边拉卷动杆边跑）的效果。

基本上要使View窗口具备滚动条，你必须做到下列事情：

定义Document大小。如果没有大小，Framework 就没有办法计算滚动条尺寸，以及卷动比例。这个大小可以是常数，也可以是个储存在每一Document 中的变量，随着执行时期变动。

- 以CScrollView 取代CView。
- 只要Document的大小改变，就将尺寸传给CScrollView 的 SetScrollSizes 函 式。如果程序设定Document为固定大小（本例就是如此），那么当然只要一开始做一次滚动条设定动作即可。
- 注意装置坐标（窗口坐标）与逻辑坐标（Document 坐标）的转换。关于此点稍后另有说明。

Application Framework 对滚动条的贡献是：

- 处理WM_HSCROLL和WM_VSCROLL消息，并相对地卷动Document（其实是移动View落在Document上的位置 ）以及移动「卷轴拉杆」（所谓的thumb）。拉杆位置可以表示出目前窗口中显示的局部在整个Document的位置。如果你按下滚动条两端箭头，卷动的幅度是一行（line），至于一行代表多少，由程序员自行决定。如果你按下的是拉杆两旁的杆子，卷动的幅度是一页（page），一页到底代表多少，也由程序员自行决定。

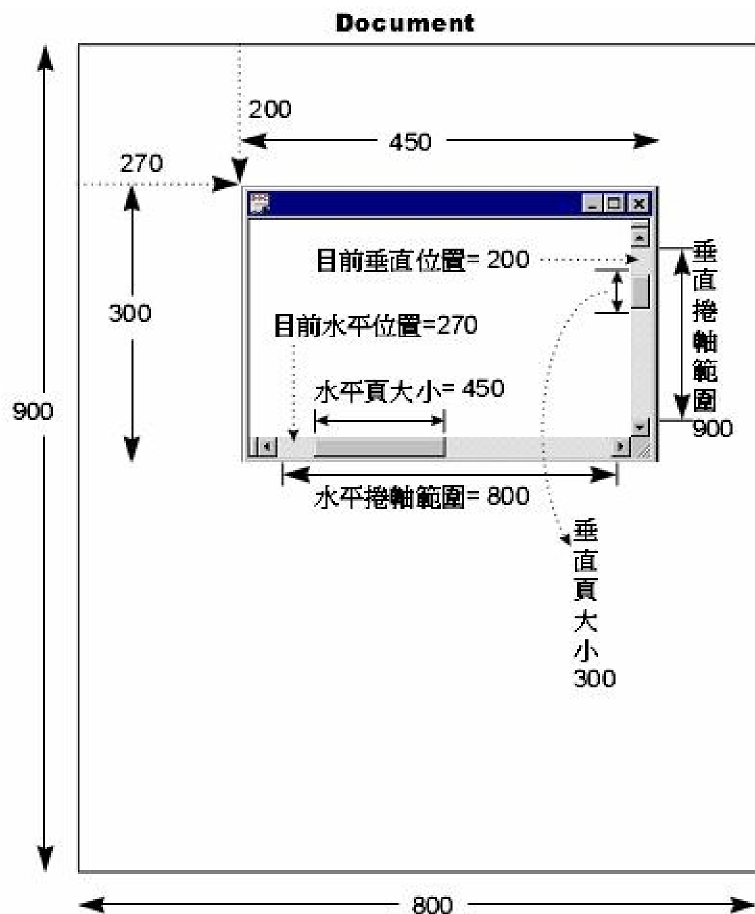
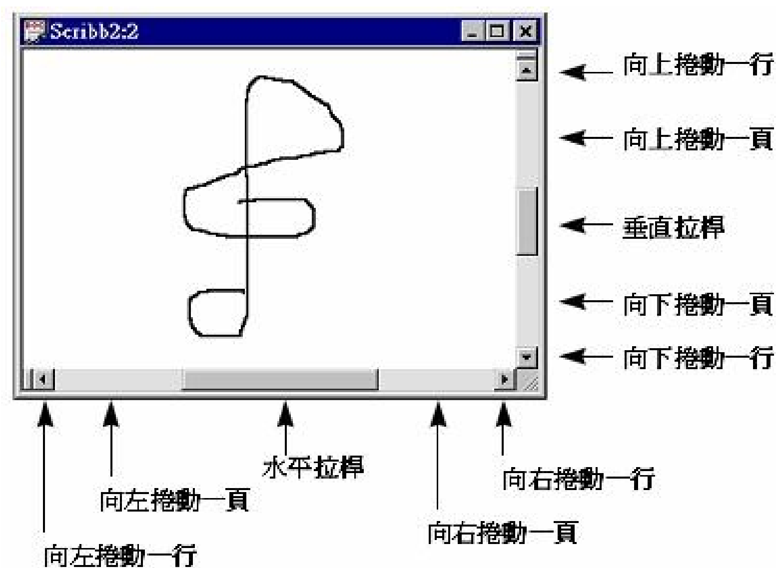


图11-5b 滚动条View 窗口与Document之间的关系



窗口一旦被放大缩小，立刻计算窗口的宽度高度与滚动条长度的比例，以重新设定卷动比例，也就是一行或一页的大小。

以下分四个步骤修改Scribble原始代码：

1 定义Document 的大小。我们的作法是设定一个变量，代表大小，并在 Document初始化时设定其值，此后全程不再改变（以简化问题）。MFC中有一个 CSize 很适合当作此一变量类型。这个成员变量在文件进行文件读写（Serialization）时也应该并入文件内容中。回忆一

下，上一章讲到笔宽时，由于每一线条有自己的一个宽度，所以笔宽资料应该在CStroke::Serialize 中读写，现在我们所讨论的文件大小却是属于整份文件的资料，所以应该在 CScribbleDoc::Serialize 中读写：

```
// in SCRIBBLEDOC.H
class CScribbleDoc : public CDocument
{
protected:
    Csize    m_sizeDoc;
    public:
    CSize GetDocSize() { return m_sizeDoc; }
    ...
};
// in SCRIBBLEDOC.CPP
void CScribbleDoc::InitDocument()
{
    m_bThickPen = FALSE;
    m_nThinWidth = 2;   // default thin pen is 2 pixels wide
    m_nThickWidth = 5;  // default thick pen is 5 pixels wide
    ReplacePen();       // initialize pen according to current width
    // 预设 Document 大小为 800 x 900 个屏幕图素
    m_sizeDoc = CSize(800,900);
}
void CScribbleDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_sizeDoc;
    }
    else
    {
        ar >> m_sizeDoc;
    }
    m_strokeList.Serialize(ar);
}
```

2 将CScribbleView 的父类由CView改变为CScrollView。同时准备改写其虚函数OnInitialUpdate，为的是稍后我们要在其中，根据Document的大小，设定卷动范围。

```
// in SCRIBBLEVIEW.H
class CScribbleView : public CScrollView
{
public:
    virtual void OnInitialUpdate();
    ...
};
// in SCRIBBLEVIEW.CPP
IMPLEMENT_DYNCREATE(CScribbleView, CScrollView)
BEGIN_MESSAGE_MAP(CScribbleView, CScrollView)
...
END_MESSAGE_MAP()
```

3 改写OnInitialUpdate，在其中设定滚动条范围。这个函数的被调用时机是在View 第一次附着到Document 但尚未显现时，由Framework 调用之。它会调用OnUpdate，不带任何Hint参数（于是lHint是0而pHint是NULL）。如果你需要做任何「只做一次」的初始化动作，而且初始化时需要 Document 的数据，那么在这里做就最合适了：

```
// in SCRIBVW.CPP
void CScribbleView::OnInitialUpdate()
{
    SetScrollSizes(MM_TEXT, GetDocument()->GetDocSize());
    // 这是 CScrollView 的成员函数。
}
```

SetScrollSizes 总共有四个参数：

- int nMapMode：代表映射模式（Mapping Mode)
- SIZE sizeTotal：代表文件大小
- const SIZE& sizePage：代表一页大小（预设是文件大小的 1/10）
- const SIZE& sizeLine：代表一行大小（预设是文件大小的 1/100）

本例的文件大小是固定的。另一种比较复杂的情况是可变大小，那么你就必须在文件大小改变之后立刻调用SetScrollSizes。

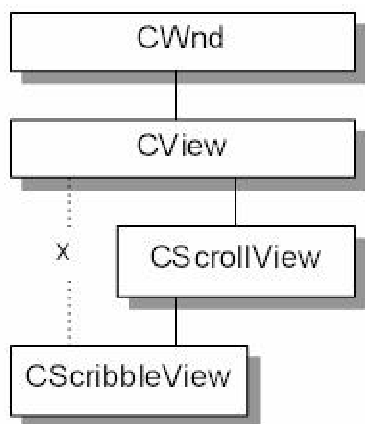
窗口上增加滚动条并不会使View的OnDraw 负担加重。我们并不因为滚动条把观察镜头移动到Document的中段或尾段，而就必须在OnDraw中重新计算绘图原点与平移向量，原因是绘图坐标与我们所使用的DC有关。当滚动条移动了DC原点，CScrollView自动会做调整，让数据的某一部份显示而某一部份隐藏。

让我做更详细的说明。「GDI 原点」是DC（注）的重要特征，许许多多CDC 成员函数的绘图结果都会受它的影响。如果我们想在绘图之前（也就是进入 OnDraw 之前）调整DC，我们可以改写虚函数OnPrepareDC，因为Framework是先调用OnPrepareDC，然后才调用OnDraw并把DC传进去。好，窗口由无滚动条到增设滚动条的过程中，之所以不必修改OnDraw 函数内容，就是因为CScrollView已经改写了CView的OnPrepareDC虚函数。Framework先调用CScrollView::OnPrepareDC再调用CScribbleView::OnDraw，所有因为滚动条而必须做的特别处理都已经在进入OnDraw之前完成了。

注意上面的叙述，别把CScrollView和CSribbleView混淆了。下图是个整理。

此类的OnPrepareDC 虚函数会因滚动条的位置而调整DC原点。

此类原针对「无滚动条窗口」而设计，所以 Step4之前的View类是直接派生自CView。彼时所写的OnDraw函数内容在如今改变了继承关系后（改继承自CScrollView），依然完全适用，原因是所有的差异性早都在进入OnDraw之前便由更早被Framework 调用的CScrollView::OnPrepare 处理掉了。



DC就是Device Context，在Windows中凡绘图动作之前一定要先获得一个 DC，它可能代表屏幕，也可能代表一个窗口，或一块内存，或打印机...。DC中有许多绘图所需的元素，包括坐标系统（映射模式）、原点、绘图工具（笔、刷、颜色...）等等。它还连接到低阶的输出装置驱动程序。由于DC，我们在程序中对屏幕作画和对打印机作画的动作才有可能完全相同。

4 修正鼠标坐标。虽说OnDraw不必因为坐标原点的变化而有任何改变，但是幕后出力的CScrollView::OnPrepareDC 却不知道什么是Windows 消息！这话看似牛头和马嘴，但我一点你就明白了。CScrollView::OnPrepareDC 虽然能够因着滚动条行为而改变GDI原点，但「改变GDI 原点」这个动作却不会影响你所接收的WM_LBUTTONDOWN 或WM_LBUTTONUP或WM_MOUSEMOVE的坐标值，原因是Windows 消息并非DC 的一个成份。因此，我们作画的基础，也就是鼠标移动产生的轨迹点坐标，必须由「以窗口绘图区左上角为原点」的窗口坐标系统，改变为「以文件左上角为原点」的逻辑坐标系统。文件中储存的，也应该是逻辑坐标。

下面是修改坐标的程序动作。其中调用的OnPrepareDC 是哪一个类的成员函数？想想看，CScribbleView派生自CScrollView，而我们并未在CscriBbleView中改写此一函数，所以程序中调用的是CScrollView::OnPrepareDC。

```
// in SCRIBVW.CPP
void CScribbleView::OnLButtonDown(UINT, CPoint point)
{
    // 由于CScrollView改变了DC原点和映射模式，所以我们必须先把
    // 装置坐标转换为逻辑坐标，再储存到Document中。
    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.DPtoLP(&point);
    m_pStrokeCur = GetDocument()->NewStroke();
    m_pStrokeCur->m_pointArray.Add(point);
    SetCapture();// 抓住鼠标
    m_ptPrev = point; // 做为直线绘图的第一个点
    return;
}
void CScribbleView::OnLButtonUp(UINT, CPoint point)
{
    ...
    if (GetCapture() != this)
    return;
    CScribbleDoc* pDoc = GetDocument();
    CClientDC dc(this);
    OnPrepareDC(&dc);  // 设定映射模式和 DC 原点
    dc.DPtoLP(&point);
    ...
}
void CScribbleView::OnMouseMove(UINT, CPoint point)
{
    ...
    if (GetCapture() != this)
    return;
    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.DPtoLP(&point);
    m_pStrokeCur->m_pointArray.Add(point);
    ...
}
```

除了上面三个函数，还有什么函数牵扯到坐标？是的，线条四周有一个外围四方形，那将在OnUpdate 中出现，也必须做坐标系统转换：

```
void CScribbleView::OnUpdate(CView* /* pSender */, LPARAM /* lHint */,
CObject* pHint)
{
    if (pHint != NULL)
    {
        if (pHint->IsKindOf(RUNTIME_CLASS(CStroke)))
        {
            //hint的确是一个CStroke对象。现在将其外围四方形设为重绘区
            CStroke* pStroke = (CStroke*)pHint;
            CClientDC dc(this);
            OnPrepareDC(&dc);
            CRect rectInvalid = pStroke->GetBoundingRect();
            dc.LPtoDP(&rectInvalid);
            InvalidateRect(&rectInvalid);
            return;
        }
    }
    // 无法识别 hint,只好假设整个画面都需重绘。
    Invalidate(TRUE);
    return;
}
```

注意，上面的LPtoDP所受参数竟然不是CPoint，而是*CRect*，那是因为 LPtoDP有重载函数（overloaded function），既可接受点，也可接受四方形。DPtoLP 也有类似的重载能力。

线条的外围四方形还可能出现在CStroke::FinishStroke 中，不过那里只是把线条数组中的点拿出来比大小，决定外围四方形罢了；而你知道，线条数组的点已经在加入时做过坐标转换了（分别在OnLButtonDown、OnMouseMove、OnLButtonUp 函数中的 AddPoint动作之前）。

至此，Document 的数据格式比起 Step1，有了大幅的变动。让我们再次分析文件文件的格式，以期获得更深入的认识与印证。我以图 11-6a 为例，共四条线段，宽度分别是 2, 5, 10, 20（十进制）。分析内容显示在图11-6b。

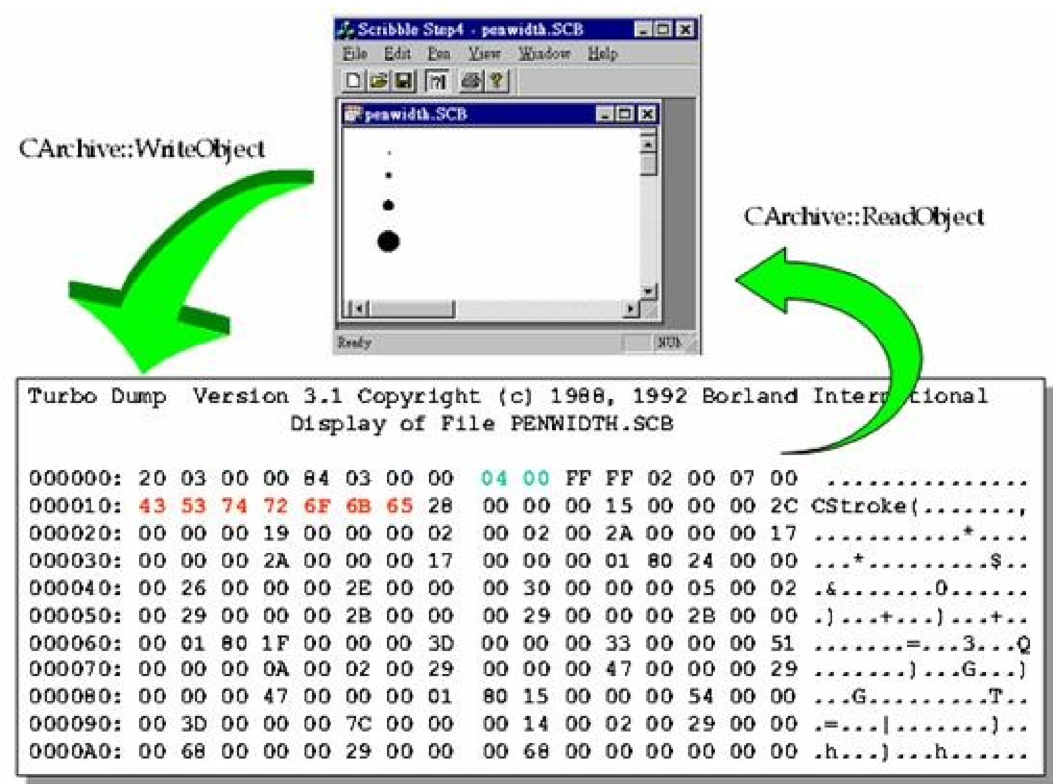


图11-6a 四条线段的图形与文件文件倾印代码

数值（hex）	说明（共173bytes）
00000320	Document 寬度（800）
00000384	Document 高度（900）
0004	表示此檔有四個 *CObList* 元素
FFFF	FFFF 亦即 -1，表示 New Class Tag
0002	Scheme no.，代表 Document 版本號碼
0007	表示後面接著的「類別名稱」有 7 個字元

43 53 74 72 6F 6B 65	類別名稱 "CStroke" 的 ASCII 碼
00000028 00000015	外圍四方形的左上角座標（膨脹一個筆寬）
0000002C 00000019	外圍四方形的右下角座標（膨脹一個筆寬）
0002	第一條線條的寬度
0002	第一條線條的點數
0000002A,00000017	第一條線條的第一個點座標
0000002A,00000017	第一條線條的第二個點座標
8001	表示接下來的物件仍舊使用舊的類別
00000024 00000026	外圍四方形的左上角座標（膨脹一個筆寬）
0000002E 00000030	外圍四方形的右下角座標（膨脹一個筆寬）
0005	第二條線條的寬度
0002	第二條線條的點數
00000029,0000002B	第二條線條的第一個點座標
00000029,0000002B	第二條線條的第二個點座標
8001	表示接下來的物件仍舊使用舊的類別
0000001F 0000003D	外圍四方形的左上角座標（膨脹一個筆寬）
00000033 00000051	外圍四方形的右下角座標（膨脹一個筆寬）
000A	第三條線條的寬度
0002	第三條線條的點數
00000029,00000047	第三條線條的第一個點座標
00000029,00000047	第三條線條的第二個點座標
8001	表示接下來的物件仍舊使用舊的類別
00000015 00000054	外圍四方形的左上角座標（膨脹一個筆寬）
0000003D 0000007C	外圍四方形的右下角座標（膨脹一個筆寬）
0014	第四條線條的寬度
0002	第四條線條的點數
00000029 00000068	第四條線條的第一個點座標
00000029 00000068	第四條線條的第二個點座標

图11-6b 文件(图11-6a)的分析

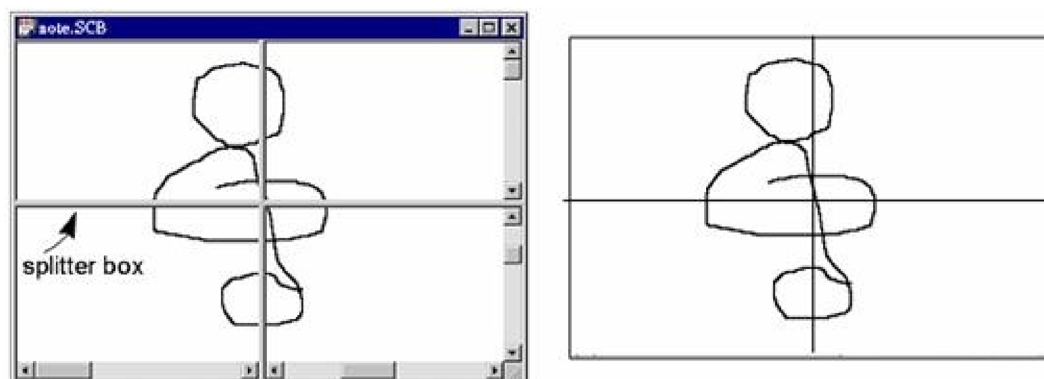
大窗口中的小窗口：Splitter

MDI程序的标准功能是允许你为同一份Document开启一个以上的Views。这种情况类似我们以多个观景窗观看同一份数据。我们可以开启任意多个 Views，各有滚动条，那么我们就可以在屏幕上同时观察一份数据的不同局部。这许多个 View 窗口各自独立运作，因此它们的观看区可能互相重迭。

如果这些隶属同一Document的Views能够结合在一个大窗口之内，又各自有独立的行为（譬如说有自己的滚动条），似乎可以带给使用者更好的感觉和更方便的使用，不是吗？

分裂窗口的功能

把View做成所谓的「分裂窗口（splitter）」是一种不错的想法。这种窗口可以分裂出数个窗口，如图11-7，每一个窗口可以映射到Document的任何位置，窗口与窗口之间彼此独立运作。



在Splitter Box 上以鼠标左键快按两下，就可以将窗口分裂开来。Splitter Box 有水平和垂直两种。分裂窗口的窗口个数，由程序而定，本例是 2x2。不同的窗口可以观察同一份 Document 的不同局部。本例虽然很巧妙地安排出一张完整的图出来，其实四个窗口各自看到原图的某一部份。

图11-7 分裂窗口（splitter window）

在Splitter Box上以鼠标左键快按两下，就可以将窗口分裂开来。Splitter Box 有水平和垂直两种。分裂窗口的窗口个数，由程序而定，本例是 2x2。不同的窗口可以观察同一份 Document 的不同局部。本例虽然很巧妙地安排出一张完整的图出来，其实四个窗口各自看到原图的某一部份。

分裂窗口的程序概念

回忆第８章所说的Document/View 架构，每次打开一个Document，需有两个窗口通力合作才能完成显示任务，一是CMDIChildWnd 窗口，负责窗口的外框架与一般行为，一是CView 窗口，负责数据的显示。但是当分裂窗口引入，这种结构被打破。现在必须有三个窗口通力合作完成显示任务（图 11-8）：

1. Document Frame 窗口：负责一般性窗口行为。其类派生自 CMDIChildWnd。

2. Splitter 窗口：负责管理各窗口。通常直接使用CSplitterWnd 类。

3. View 窗口：负责数据的显示。其类派生自CView。

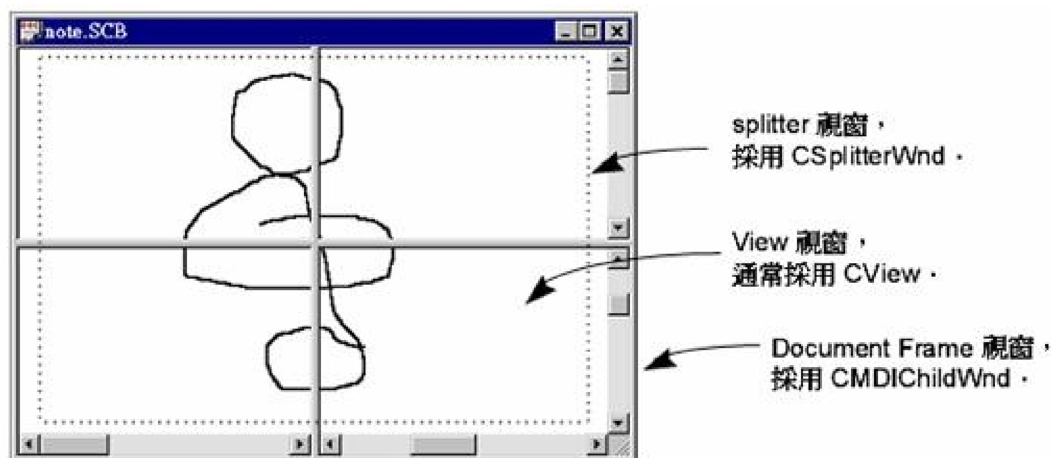


图11-8 欲使用分裂窗口，必须三个对象合作才能完成显示任务，

一是Document Frame 窗口，负责一般性窗口行为；二是CSplitterWnd窗口，管理窗口内部空间（各个窗口）；三是CView窗口，负责显示数据

给SDK程序员

你有以SDK撰写MDI程序的经验吗？MDI程序有三层窗口架构：

程序员想要控制MDI Child 窗口的大小、位置、排列状态，必须藉助另一个已经由Windows 系统定义好的窗口，此窗口称为MDI Client 窗口，其类 名称为"MDICLIENT"。

Frame窗口、Client窗口和Child窗口构成MDI的三层架构。Frame窗口产生之后，通常在WM_CREATE时机就以CreateWindow("MDICLIENT",...）;的方式建立Client窗口，此后几乎所有对Child窗口的管理工作，诸如产生新的Child 窗口、重新排列窗口、重新排列图示、在选单上列出已开启窗口...等等，都由 Client代劳，只要Frame窗口向Client 窗口下命令（送 MDI 消息如 WM_MDICREATE或WM_MDITILE 就去）即可。

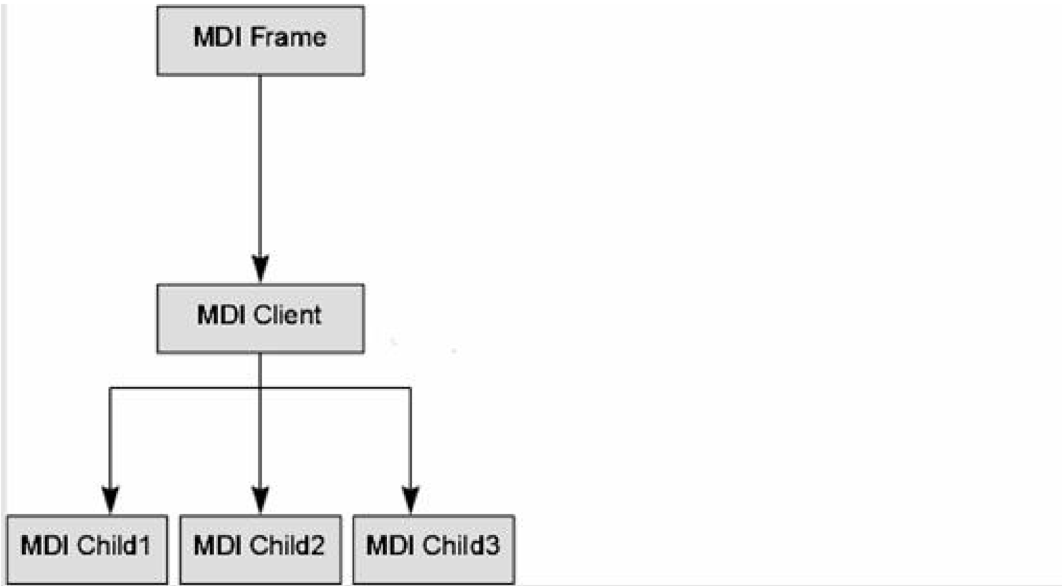
你可以把CSplitterWnd对象视为MDI Client，观念上比较容易打通。

在 SDK 程序中，MDI Child 窗口的消息预设处理函数是DefMDIChildProc()，而不是DefWindowProc()。

MDI Client 是 Windows 预设好的窗口类，名为"MDIClient"。你也可以把它视为一种控制组件。

MDI Frame窗口发出 MDI 消息（如 WM_MDICASCADE、WM_MDITILE），命令 MDI Client 窗口管理其子窗口（管理动作包括窗口产生、位置排列等等)。

在 SDK 程序中，MDI Frame 窗口的消息预设处理函数是DefFrameProc()，而不是DefWindowProc()。



分裂窗口之实现

让我先把Scribble目前使用的类之中凡与本节主题有关的，做个整理。

Visual C++ 4.0 以前的版本，AppWizard为Scribble产生的类是这样子的：

用途	类名称	基类（MFC 类）
main frame	CMainFrame	CMDIFrameWnd
document frame	直接使用 MFC 类	CMDIChildWnd CMDIChildWnd
view	CScribbleView	CView
document	CscribbleDoc	CDocument

而其CMultiDocTemplate对象是这样子的：

```
pDocTemplate = new CMultiDocTemplate(
IDR_SCRIBTYPE,
RUNTIME_CLASS(CScribbleDoc),
RUNTIME_CLASS(CMDIChildWnd),
RUNTIME_CLASS(CScribbleView));
```

为了加上分裂窗口，我们必须利用ClassWizard 新增一个类（在 Scribble 程序中名为CScribbleFrame），派生自CMDIChildWnd，并让它拥有一个CSplitterWnd 对象，名为m_wndSplitter。然后为CSrcibbleFrame 改写 OnCreateClient 虚函数，在其中调用m_wndSplitter.Create 以产生分裂窗口、设定窗口个数、设定窗口的最初尺寸等初始状态。最后，当然，我们不能够再直接以CMDIChildWnd负责document frame窗口，而必须以CScribbleFrame取代之。也就是说，得改变CMultiDocTemplate 构造函数的第三个参数：

```
pDocTemplate = new CMultiDocTemplate(
IDR_SCRIBTYPE,
RUNTIME_CLASS(CScribbleDoc),
RUNTIME_CLASS(CScribbleFrame),
RUNTIME_CLASS(CScribbleView));
```

俱往矣！Visual C++ 4.0之后的AppWizard为Scribble产生的类是这个样子：

用途	类名称	基类
main frame	*CMainFrame*	*CMDIFrameWnd*
document frame	*CChildFrame*	*CMDIChildWnd*
view	*CScribbleView*	*CView*
document	*CScribbleDoc*	*CDocument*

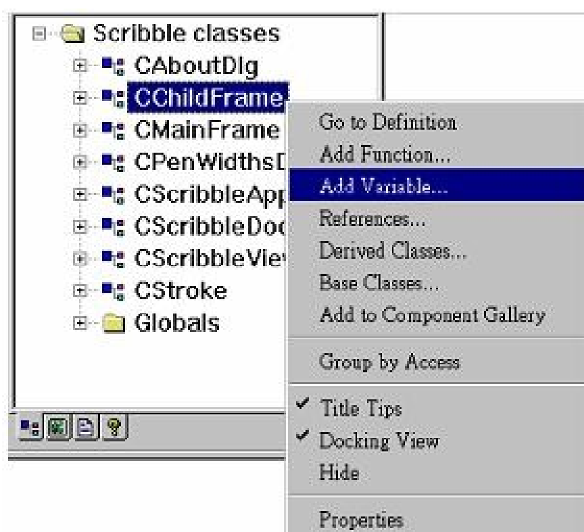
而其CMultiDocTemplate对象是这样子的：

```
pDocTemplate = new CMultiDocTemplate(
IDR_SCRIBTYPE,
RUNTIME_CLASS(CScribbleDoc),
RUNTIME_CLASS(CChildFrame),
RUNTIME_CLASS(CScribbleView));
```

这就方便多了，CChildFrame相当于以前（MFC 4.0之前）你得自力完成的CScribbleFrame。现在，我们可以从「为此类新添成员变量」开始作为。

以下是加上分裂窗口的步骤：

- 在ClassView（注意，不是ClassWizard）中选择CChildFrame。按下右键，选择突冒式选单中的【Add Variable】



- 出现【Add Member Variable】对话框。填充如下，然后选按【OK】。



现在你可以从ClassView 画面中实时看到CChildFrame 的新变量。

- 打开ChildFrm.cpp，在izardBar的【Messages】清单中选择 OnCreateClient。
- 以Yes回答WizardBar 的询问，产生处理例程。
- 在函数空壳中键入以下内容：

```
return m_wndSplitter.Create(this, 2, 2, CSize(10, 10), pContext);
```

- 回到ClassView 之中，你可以看到新的函数。

CSplitterWnd::Create 正是产生分裂窗口的关键，它有七个参数：

1. 表示父窗口。这里的this代表的是CChildFrame窗口。

2. 分裂窗口的水平窗口数（row）

3. 分裂窗口的垂直窗口数（column）

4. 窗口的最小尺寸（应该是一个CSize 对象）

5. 在窗口上使用哪一个View类 。此参数直接取用Framework交给OnCreateClient的第二个参数即可。

6.指定分裂窗口的风格。预设值是：`WS_CHILD|WS_VISIBLE|WS_HSCROLL| WS_VSCROLL|SPLS_DYNAMIC_SPLIT`，意思就是一个可见的子窗口，有着水平卷轴和垂直滚动条，并支持动态分裂。关于动态分裂（以及所谓的静态分裂），第13 章将另有说明。

7.分裂窗口的ID。默认值是AFX_IDW_PANE_FIRST，这将成为第一个窗口的 ID。

我们的原始代码有了下列变化：

```
// in CHILDFRM.H
class CChildFrame : public CMDIChildWnd
{
    protected:
    CSplitterWnd m_wndSplitter;
    protected:
    virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext);
    ...
};
// in CHILDFRM.CPP
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT /* lpcs */,
CCreateContext* pContext
{
    return m_wndSplitter.Create(this,
    2, 2,           //TODO: adjust the number of rows, columns
    CSize(10, 10),  //TODO: adjust the minimum pane size
    pContext);
}
```

本章回顾

这一章里我们追求的是精致化。

Scribble Step3已经有绘图、文件读写、变化笔宽的基本功能，但是「连接到同一份Document 的不同的Views」之间却不能够做到同步更新的视觉效果，此外View窗口中没有滚动条也是遗憾的事情。

Scribble Step4弥补了上述遗憾。它让「连接到同一份Document 的不同的 Views」之间做到同步更新——关键在于CDocument::UpdateAllViews和 CView::Update 两个虚函数。而由于同步更新引发的绘图效率问题，所以我们又学会了如何设计所谓的hint，让绘图动作更聪敏些。也因为hint缘故，我们改变了Document的格式，为每一线条加上一个外围四方形记录。

在滚动条方面，MFC 提供了一个名为 CScrollView 的类，内有滚动条功能，因此直接拿来用就好了。我们唯一要担心的是，从 CView 改为 CScrollView，原先的 OnDraw 绘图动作要不要修改？毕竟，卷来卷去把原点都不知卷到哪里去了，何况还有映射模式（坐标系统）的问题。这一点是甭担心了，因为application framework 在调用OnDraw 之前，已经先调用了OnPrepareDC，把问题解决掉了。唯一要注意的是，送进 OnDraw 的滑鼠坐标点应该改为逻辑坐标，以文件左上角为原点。DP2LP 函数可以帮我们这个忙。

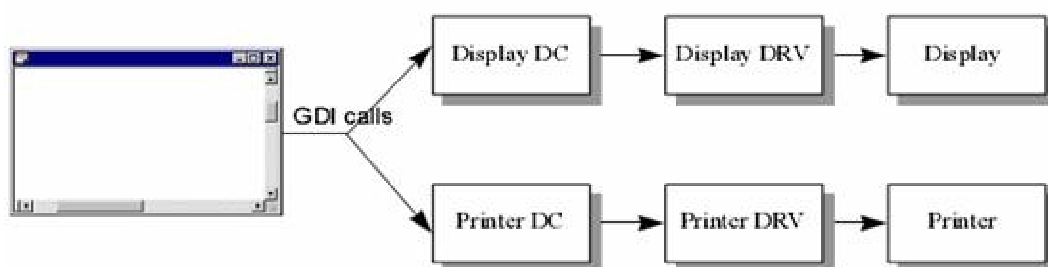
此外，我们接触了另一种新而且更精致的UI接口：分裂窗口，让一个窗口分裂为数个窗口，每一个窗口容纳一个View。MFC提供CSplitterWnd 做此服务。

第12章 打印与预览

「打印」绝对是个大工程，「打印预览」是个更大的工程。如果你是一位 SDK 程序员，而你分配到的工作是为公司的绘图软件写一个印前预浏系统，那么我真的替你感到忧郁。可如果你使用 MFC，情况又大不相同了。

概观

Windows 的DC观念，在程序的绘图动作与实际设备的驱动程序之间做了一道隔离，使得绘图动作完全不需修改就可以输出到不同的设备上：



即便如此，打印仍然有其琐碎的工作需要由程序员承担。举个例子，屏幕窗口有卷动杆，打印机没有，于是「分页」就成了一门学问。另外，如何中断打印？如何设计水平方向（landscape）或垂直方向（portrait）的打印输出？

landscape，风景画，代表横向打印；portrait，人物画，代表纵向打印。

如果曾经有过SDK程序经验，你一定知道，把数据输出到屏幕上和输出到打印机上几乎是相同的一件事，只要换个DC（注）就好了。MFC 甚至不要求程序员的任何动作，即自动供应打印功能和预览功能。拿前面各版本的Scribble为例，我们可曾为了输出任何东西到打印机上而特别考虑什么程序代码？完全没有！但它的确已拥有打印和预览功能，你不妨执行 Step4 的【File/Print...】以及【File/Print Preview】看看，结果如图 12-1a。

注：DC 就是Device Context，在Windows中凡绘图动作之前一定要先获得一个DC，它可能代表全屏幕，也可能代表一个窗口，或一块内存，或打印机...。DC中有许多绘图所需的元素，包括坐标系统（映射模式）、原点、绘图工具（笔、刷、颜色...）等等。它还连接到低阶的输出装置驱动程序。由于 DC，我们在程序中对屏幕作画和对打印机作画的动作才有可能完全相同。

Scribble 程序之所以不费吹灰之力即拥有打印与预览功能，是因为负责数据显示的 CSribbleView::OnDraw 函数接受了一个DC参数，此DC如果是 display DC，所有的输出就往屏幕送，如果是printer DC，所有输出就往打印机送。至于OnDraw 到底收到什么样的 DC，则由Framework决定——想当然耳Framework 会依使用者的动作决定之。

MFC把整个打印机制和预览机制都埋在application framework 之中了，我们因此也有了标准的UI界面可以使用，如标准的【打印】对话框、【打印设定】对话框、【打印中】对话框等等，请看图 12-1。

我将在这一章介绍 MFC 的印表与预览机制，以及如何强化它。

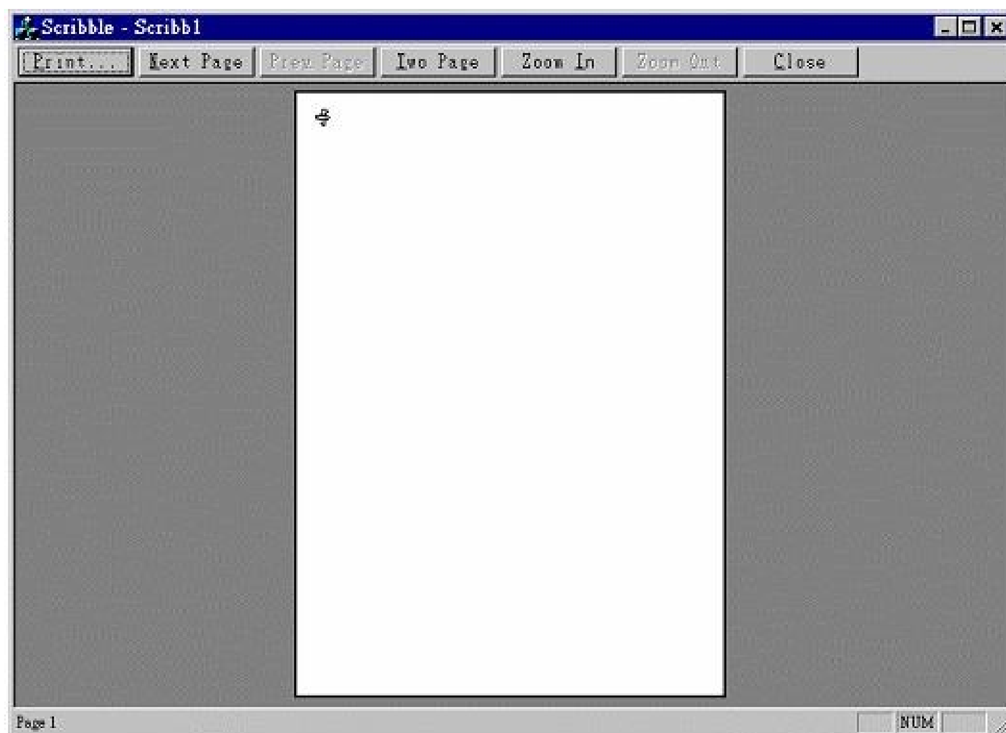


图12-1a 不需考虑任何与打印相关的程序动作，Scribble 即已具备印表与预览功能（只要我们一开始在AppWizard 的步骤四对话框中选择【 Printing and Print Preview】 项目）。打印出来的图形大小并不符合理想，从预览画面中就可察知。这正是本章要改善的地方。



图12-1b 标准的打印UI接口。本图是选按Scribble 的【 File/Print...】命令项之后获得的【打印】对话框。



图12-1c 你可以选按Scribble 的【File/Print Setup...】命令项，

获得设定打印机的机会。

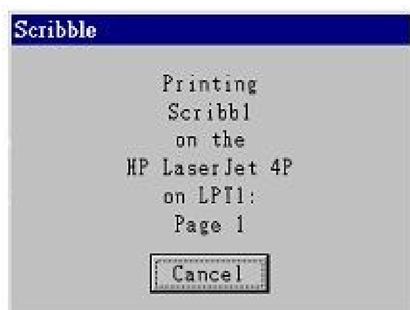


图12-1d 打印过程中会出现一个标准的【打印状态】对话框，

允许使用者中断打印动作。

Scribble Step5加强了印表功能以及预览功能。MFC 各现成类之中已有印表和预览机制，我要解释的是它的运作模式、执行效果、以及改善之道。图 12-2 就是 Scribble Step5的预览效果，UI 方面并没有什么新东西，主要的改善是，图形的输出大小比较能够被接受了，每一份文件并且分为两页，第一页是文件名称（文件名称），第二页才是真正的文件内容，上有一表头。

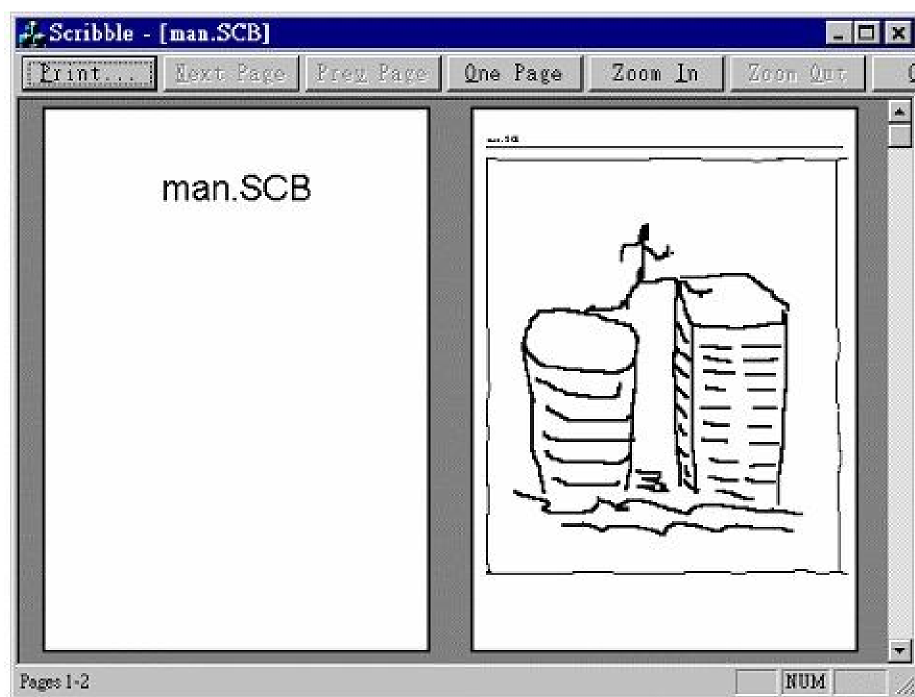


图12-2 Scribble Step5打印预览。第一页是文件名称，第二页是文件内容

打印动作的后台原理

开始介绍MFC的打印机制之前，我想，如果先让你了解打印的背后原理，可以帮助你掌握其本质。

Windows 的所有绘图指令，都集中在 GDI 模块之中，称为 GDI 绘图函数，例如：

```
TextOut(hPr, 50, 50, szText, strlen(szText)); // 输出一字符串
Rectangle(hPr, 10, 10, 50, 40); // 画一个四方形
Ellipse(hPr, 200, 50, 250, 80); // 画一个椭圆形
Pie(hPr, 350, 50, 400, 100, 400, 50, 400, 100);  // 画一个圆饼图
MoveTo(hPr, 50, 100); // 将画笔移动到新位置
LineTo(hPr, 400, 50); // 从前一位置画直线到新位置
```

图形输往何方？关键在于DC，这是任何GDI绘图函数的第一个参数，可以是GetDC或BeginPaint函数所获得的「显示屏DC」（以下是SDK程序写法）：

```
HDC hDC;
PAINTSTRUCT ps; // paint structure
hDC = BeginPaint(hWnd, &ps);
```

也可以是利用 CreateDC 获得的一个「打印机DC」：

```
HDC hPr;
hPr=CreateDC(lpPrintDriver,lpPrintType,lpPrintPort,(LPSTR)NULL);
```

其中前三个参数分别是与打印机有关的信息字符串，可以从WIN.INI的【windows】section中获得，各以逗号分隔，例如：

```
device=HP LaserJet 4P/4MP,HPPCL5E,LPT1:
```

代表三项意义：

- Print Driver = HP LaserJet 4P/4MP
- Print Type = HPPCL5E
- Print Port = LPT1:

SDK 程序中对于打印所需做的努力，最低限度到此为止。显然，困难度并不高，但是其中尚未参杂对打印机的控制，而那是比较麻烦的事儿。换句话说我们还得考虑「分页」的问题。以文字为例，我们必须取得一页（一张纸）的大小，以及字形的高度，从而计算扣除留白部份之后，一页可容纳几行：

```
TEXTMETRIC TextMetric;
int LineSpace;
int nPageSize;
int LinesPerPage;
GetTextMetrics(hPr, &TextMetric);  // 取得字形数据
LineSpace=TextMetric.tmHeight+TextMetric.tmExternalLeading;//计算字高
nPageSize = GetDeviceCaps(hPr, VERTRES);  // 取得纸张大小
LinesPerPage = nPageSize / LineSpace - 1; // 一页容纳多少行
```

然后再以循环将每一行文字送往打印机：

```
Escape(hPr, STARTDOC, 4, "PrntFile text", (LPSTR) NULL);
CurrentLine = 1;
for (...) {
    ...//取得一行文字，放在char pLine[128]中，长度为LineLength。
    TextOut(hPr, 0, CurrentLine*LineSpace, (LPSTR)pLine, LineLength);
    if (++CurrentLine > LinesPerPage ) {
        CurrentLine = 1;   // 重设行号
        IOStatus = Escape(hPr, NEWFRAME, 0, 0L, 0L);  // 换页
        if (IOStatus < 0 || bAbort)
        break;
    }
}
if (IOStatus >= 0 && !bAbort) {
    Escape(hPr, NEWFRAME, 0, 0L, 0L);
    Escape(hPr, ENDDOC, 0, 0L, 0L);
}
```

其中的Escape用来传送命令给打印机（打印机命令一般称为escape code），它是一个Windows API 函数。

打印过程中我们还应该提供一个中断机制给使用者。Modeless 对话框可以完成此一使命，我们可以让它出现在打印过程之中。这个对话框应该在打印程序开始之前先做起来，外形类似图 12-1d：

```
HWND hPrintingDlgWnd;  // 这就是【Printing】对话框
FARPROC lpPrintingDlg; //【Printing】对话框的窗口函数
lpPrintingDlg=MakeProcInstance(PrintingDlg, hInst);
hPrintingDlgWnd=CreateDialog(hInst,"PrintingDlg",hWnd,lpPrintingDlg);
ShowWindow (hPrintingDlgWnd, SW_NORMAL);
```

负责此一中断机制的对话框函数很简单，只检查【OK】钮有没有被按下，并据以改变bAbort的值：

```
int FAR PASCAL PrintingDlg(HWND hDlg,unsigned msg,WORD wParam,LONG lParam)
{
    switch(msg) {
        case WM_COMMAND:
            return (bAbort = TRUE);
        case WM_INITDIALOG:
            SetFocus(GetDlgItem(hDlg, IDCANCEL));
            SetDlgItemText(hDlg, IDC_FILENAME, FileName);
            return (TRUE);
        }
    return (FALSE);
}
```

从应用程序的眼光来看，这样就差不多了。然而数据真正送到打印机上，还有一大段曲折过程。每一个送往打印机 DC 的绘图动作，其实都只被记录为 metafile（注）储存在你的TEMP 目录中。当你调用Escape(hPr, NEWFRAME, ...)，打印机驱动程序（.DRV）会把这些metafile转换为打印机语言（control sequence 或Postscript），然后通知GDI模组，由GDI把它储存为 ~SPL 文件，也放在TEMP 目录中，并删除对应之metafile。之后，GDI模块再送出消息给打印管理器 Print Manager，由后者调用OpenComm、WriteComm 等低阶通讯函数（也都是 Windows API 函数），把打印机命令传给打印机。整个流程请参考图 12-3。

注：metafile也是一种图形记录规格，但它记录的是绘图动作，不像 bitmap 记录的是真正的图形数据。所以播放metafile比播放bitmap 慢，因为多了一层绘图函数解读动作；但它的大小比 bitmap 小很多。用在有许多四形、圆形、工程几何图形上最为方便。

这个曲折过程之中就产生了一个问题。~SPL 这种文件很大，如果你的TEMP 目录空间不够充裕，怎么办？如果Printer Manager把积存的~SPL内容消化掉后能够空出足够磁碟空间的话，那么GDI模块就可以下命令（送消息）给 Printer Manager，先把积存的~SPL 文件处理掉。问题是，在Windows 3.x之中，我们的程序此刻正忙着做绘图动作，GDI 没有机会送消息给Printer Manager（因为Windows 3.x是个非强制性多任务系统)。解决方法是你先准备一个callback 函数，名称随你取，通常名为AbortProc：

```
FARPROC lpAbortProc;
lpAbortProc = MakeProcInstance(AbortProc, hInst);
Escape(hPr, SETABORTPROC, NULL, (LPSTR)(long)lpAbortProc, (LPSTR)NULL);
```

GDI模块在执行Escape(hPr, NEWFRAME...) 的过程中会持续调用这个 callback 函数，想办法让你的程序释放出控制权：

```
int FAR PASCAL AbortProc(hDC hPr, int Code)
{
    MSG msg;
    while (!bAbort && PeekMessage(&msg, NULL, NULL, NULL, TRUE))
    if (!IsDialogMessage(hAbortDlgWnd, &msg)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (!bAbort);
}
```

你可以从 VC++ 4.0 所附的这个范例程序获得有关打印的极佳实例：

```
\MSDEV\SAMPLES\SDK\WIN32\PRINTER
```

也可以在Charles Petzold所著的Programming Windows 3.1第15章，或是其新版Programming Windows 95第15章，获得更深入的数据。

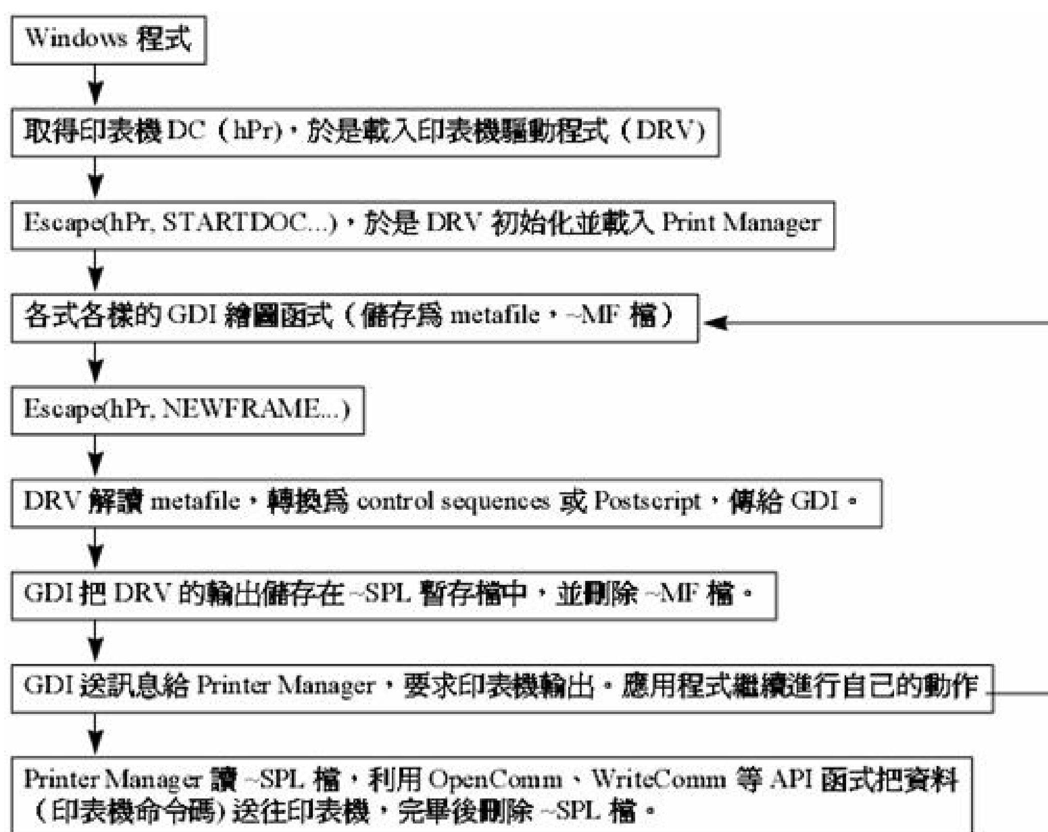


图12-3

Windows 程序的打印机输出动作详解

以下就是SDK 程序中有关打印程序的一个实际片段。

```
#01 hSaveCursor = SetCursor(hHourGlass); // 把滑鼠游標設爲砂漏狀
#02 hPr = CreateDC("HP LaserJet 4P/4MP", "HPPCL5E", "LPT1:", (LPSTR) NULL);
#03
#04 // 設定 AbortProc callback 函式
#05 lpAbortProc = MakeProcInstance(AbortProc, hInst);
#06 Escape(hPr, SETABORTPROC, NULL, (LPSTR) (long) lpAbortProc, (LPSTR) NULL);
#07 bAbort = FALSE;
#08
#09 Escape(hPr, STARTDOC, 4, "PrntFile text", (LPSTR) NULL);
#10
#11 // 設定 Printing 對話盒及其視窗函式
#12 lpPrintingDlg =  MakeProcInstance(PrintingDlg, hInst);
#13 hPrintingDlgWnd = CreateDialog(hInst, "PrintingDlg", hWnd, lpPrintingDlg);
#14 ShowWindow (hPrintingDlgWnd, SW_NORMAL);
#15 EnableWindow(hWnd, FALSE); // 令其父視窗 (也就是程式的主視窗) 除能
#16 SetCursor(hSaveCursor);     // 滑鼠游標形狀還原
#17
#18 GetTextMetrics(hPr, &TextMetric);
#19 LineSpace = TextMetric.tmHeight + TextMetric.tmExternalLeading;
#20 nPageSize = GetDeviceCaps (hPr, VERTRES);
#21 LinesPerPage = nPageSize / LineSpace - 1;
#22 dwLines = SendMessage(hEditWnd, EM_GETLINECOUNT, 0, 0L);
#23 CurrentLine = 1;
#24
#25 for (dwIndex = IOStatus = 0; dwIndex < dwLines; dwIndex++) {
#26     pLine[0] = 128;
#27     pLine[1] = 0;
#28     LineLength = SendMessage(hEditWnd, EM_GETLINE,
#29         (WORD)dwIndex, (LONG)((LPSTR)pLine));
#30     TextOut(hPr, 0, CurrentLine*LineSpace, (LPSTR)pLine, LineLength);
#31     if (++CurrentLine > LinesPerPage ) {
#32         CurrentLine = 1;
#33         IOStatus = Escape(hPr, NEWFRAME, 0, 0L, 0L);
#34         if (IOStatus<0 || bAbort)
#35             break;
#36     }
#37 }
#38
#39 if (IOStatus >= 0 && !bAbort) {
#40     Escape(hPr, NEWFRAME, 0, 0L, 0L);
#41     Escape(hPr, ENDDOC, 0, 0L, 0L);
#42 }
#43
#44 EnableWindow(hWnd, TRUE);
#45
#46 DestroyWindow(hPrintingDlgWnd);
#47 FreeProcInstance(lpPrintingDlg);
#48 FreeProcInstance(lpAbortProc);
#49 DeleteDC(hPr);
```

上述各个Escape调用，是在Windows 3.0 下的传统作法，在Windows 3.1以及 Win32 之中有对应的 API 函数如下：

Windows 3.0 作法	Windows 3.1 作法
Escape(hPr, SETABORTPROC, ...)	SetAbortProc(HDC hdc, ABORTPROC lpAbortProc)
Escape(hPr, STARTDOC, ...)	StartDoc(HDC hdc, CONST DOCINFO* lpdi)
Escape(hPr, NEWFRAME, ...)	EndPage(HDC hdc)
Escape(hPr, ENDDOC, ...)	EndDoc(HDC hdc)

MFC 预设的打印机制

好啦，关于打印，其实有许多一成不变的动作！为什么开发工具不帮我们做掉呢？好比说，从WIN.INI中取得目前打印机的数据然后利用CreateDC取得打印机DC，又好比说设计标准的【打印中】对话框，以及标准的打印中断函数 AbortProc。

事实上MFC的确已经帮我们做掉了一大部份的工作。MFC已内含打印机制，那么将Framework 整个纳入EXE文件中的你当然也就不费吹灰之力得到了印表功能。只要OnDraw函数设计好了，不但可以在屏幕上显示数据，也可以在打印机上显示数据。有什么是我们要负担的？没有了！Framework 传给OnDraw一个 DC，视情况的不同这个DC可能是显示屏DC，也可能是打印机DC，而你知道，Windows 程序中的图形输出对象完全取决于DC：

- 当你改变窗口大小，产生WM_PAINT，OnDraw 会收到一个「显示屏DC」。
- 当你选按【File/Print...】，OnDraw 会收到一个「打印机DC」。

数章之前讨论 CView 时我曾经提过，OnDraw是CView类中最重要的成员函数，所有的绘图动作都应该放在其中。请注意，OnDraw 接受一个「CDC 对象指针」做为它的参数。当窗口接受 WM_PAINT 消息，Framework 就调用OnDraw 并把一个「显示屏DC」传过去，于是OnDraw输出到屏幕上。

Windows 的图形装置接口(GDI）完全与硬件无关，相同的绘图动作如果送到「显示屏DC」，就是在屏幕上绘图，如果送到「打印机DC」，就是在打印机上绘图。这个道理很容易就解释了为什么您的程序代码没有任何特殊动作却具备印表功能：当使用者按下【File/Print】，application framework送给OnDraw的是一个「打印机 DC」而不再是「显示幕DC」。

在 MFC 应用程序中，View 和 application framework 分工合力完成印表工作。Application framework 的责任是：

- 显示【Print】对话框，如图12-1b。
- 为打印机产生一个 CDC 对象。
- 调用 CDC对象的StartDoc和EndDoc两函数。
- 持续不断地调用CDC对象的StartPage，通知View应该输出哪一页；一页打印完毕则调用CDC对象的EndPage。

我们（程序员）在View对象上的责任是：

- 通知application framework总共有多少页要打印。
- application framework 要求打印某特定页时，我们必须将Document中对应的部份输出到打印机上。
- 配置或释放任何GDI资源，包括笔、刷、字形...等等。

- 如果需要，送出任何escape 代码改变打印机状态，例如走纸、改变打印方向等等。

送出escape 代码的方式是，调用CDC对象的Escape 函数。

现在让我们看看这两组工作如何交叉在一起。为实现上述各项交互动作，CView 定义了几个相关的成员函数，当你在 AppWizard 中选择【Printing and Print Preview】选项之后，除了OnDraw，你的 View 类内还被加入了三个虚函数空壳：

```
// in SCRIBBLEVIEW.H
class CScribbleView : public CScrollView
{
    ...
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    ...
};
// in SCRIBBLEVIEW.CPP
BOOL CScribbleView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}
void CScribbleView::OnBeginPrinting(CDC* /*pDC*/,CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}
void CScribbleView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}
```

改写这些函数有助于我们在framework的打印机制与应用程序的View对象之间架起沟通桥梁。

为了了解MFC中的打印机制，我又动用了我的法宝：Visual C++ Debugger。我发现，AppWizard 为我的View做出这样的Message Map：

```
BEGIN_MESSAGE_MAP(CScribbleView, CScrollView)
...
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

显然，当【File/Print...】被按下，命令消息将流往CView::OnFilePrint 去处理，于是我以Debugger进入该位置并且一步一步执行，得到图12-4的结果。

```
// in VIEWPRNT.CPP
#0001  void CView::OnFilePrint()
#0002  {
#0003      // get default print info
#0004  ❶  CPrintInfo printInfo;
#0005      ASSERT(printInfo.m_pPD != NULL);    // must be set
#0006
#0007      if (GetCurrentMessage()->wParam == ID_FILE_PRINT_DIRECT)
#0008      {
#0009          CCommandLineInfo* pCmdInfo = AfxGetApp()->m_pCmdInfo;
#0010
#0011          if (pCmdInfo != NULL)
#0012          {
#0013  ❷          if (pCmdInfo->m_nShellCommand == CCommandLineInfo::FilePrintTo)
#0014              {
#0015                  printInfo.m_pPD->m_pd.hDC = ::CreateDC(pCmdInfo->m_strDriverName,
#0016                      pCmdInfo->m_strPrinterName, pCmdInfo->m_strPortName, NULL);
#0017                  if (printInfo.m_pPD->m_pd.hDC == NULL)
#0018                  {
#0019                      AfxMessageBox(AFX_IDP_FAILED_TO_START_PRINT);
#0020                      return;
#0021                  }
#0022              }
#0023          }
#0024
#0025          printInfo.m_bDirect = TRUE;
#0026      }
#0027
#0028  ❸  if (OnPreparePrinting(&printInfo))
#0029      {
#0030          // hDC must be set (did you remember to call DoPreparePrinting?)
#0031          ASSERT(printInfo.m_pPD->m_pd.hDC != NULL);
#0032
#0033  ❹      // gather file to print to if print-to-file selected
#0034          CString strOutput;
#0035          if (printInfo.m_pPD->m_pd.Flags & PD_PRINTTOFILE)
#0036          {
#0037              // construct CFileDialog for browsing
#0038              CString strDef(MAKEINTRESOURCE(AFX_IDS_PRINTDEFAULTEXT));
#0039              CString strPrintDef(MAKEINTRESOURCE(AFX_IDS_PRINTDEFAULT));
#0040              CString strFilter(MAKEINTRESOURCE(AFX_IDS_PRINTFILTER));
#0041              CString strCaption(MAKEINTRESOURCE(AFX_IDS_PRINTCAPTION));
#0042              CFileDialog dlg(FALSE, strDef, strPrintDef,
#0043                  OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT, strFilter);
#0044              dlg.m_ofn.lpstrTitle = strCaption;
#0045
```

```
#0046            if (dlg.DoModal() != IDOK)
#0047                return;
#0048
#0049            // set output device to resulting path name
#0050            strOutput = dlg.GetPathName();
#0051        }
#0052
#0053  ❺    // set up document info and start the document printing process
#0054        CString strTitle;
#0055        CDocument* pDoc = GetDocument();
#0056        if (pDoc != NULL)
#0057                strTitle = pDoc->GetTitle();
#0058        else
#0059                GetParentFrame()->GetWindowText(strTitle);
#0060        if (strTitle.GetLength() > 31)
#0061                strTitle.ReleaseBuffer(31);
#0062        DOCINFO docInfo;
#0063        memset(&docInfo, 0, sizeof(DOCINFO));
#0064        docInfo.cbSize = sizeof(DOCINFO);
#0065        docInfo.lpszDocName = strTitle;
#0066        CString strPortName;
#0067        int nFormatID;
#0068        if (strOutput.IsEmpty())
#0069        {
#0070            docInfo.lpszOutput = NULL;
#0071            strPortName = printInfo.m_pPD->GetPortName();
#0072            nFormatID = AFX_IDS_PRINTONPORT;
#0073        }
#0074        else
#0075        {
#0076            docInfo.lpszOutput = strOutput;
#0077            AfxGetFileTitle(strOutput,
#0078                strPortName.GetBuffer(_MAX_PATH), _MAX_PATH);
#0079            nFormatID = AFX_IDS_PRINTTOFILE;
#0080        }
#0081
#0082  ❻    // setup the printing DC
#0083        CDC dcPrint;
#0084        dcPrint.Attach(printInfo.m_pPD->m_pd.hDC);  // attach printer dc
#0085        dcPrint.m_bPrinting = TRUE;
#0086  ❼    OnBeginPrinting(&dcPrint, &printInfo);
#0087  ❽    dcPrint.SetAbortProc(_AfxAbortProc);
#0088
#0089        // disable main window while printing & init printing status dialog
#0090  ❾    AfxGetMainWnd()->EnableWindow(FALSE);
#0091        CPrintingDialog dlgPrintStatus(this);
#0092
#0093        CString strTemp;
#0094        dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_DOCNAME, strTitle);
#0095
#0096        dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_PRINTERNAME,
#0097            printInfo.m_pPD->GetDeviceName());
#0098        AfxFormatString1(strTemp, nFormatID, strPortName);
#0099        dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_PORTNAME, strTemp);
#0100
#0101        dlgPrintStatus.ShowWindow(SW_SHOW);
#0102        dlgPrintStatus.UpdateWindow();
#0103
#0104        // start document printing process
#0105  ❿    if (dcPrint.StartDoc(&docInfo) == SP_ERROR)
```

```
#0106         {
#0107            // enable main window before proceeding
#0108            AfxGetMainWnd()->EnableWindow(TRUE);
#0109
#0110            // cleanup and show error message
#0111            OnEndPrinting(&dcPrint, &printInfo);
#0112            dlgPrintStatus.DestroyWindow();
#0113            dcPrint.Detach();   // will be cleaned up by CPrintInfo destructor
#0114            AfxMessageBox(AFX_IDP_FAILED_TO_START_PRINT);
#0115            return;
#0116         }
#0117
#0118         // Guarantee values are in the valid range
#0119         UINT nEndPage = printInfo.GetToPage();
#0120         UINT nStartPage = printInfo.GetFromPage();
#0121
#0122         if (nEndPage < printInfo.GetMinPage())
#0123            nEndPage = printInfo.GetMinPage();
#0124         if (nEndPage > printInfo.GetMaxPage())
#0125            nEndPage = printInfo.GetMaxPage();
#0126
#0127         if (nStartPage < printInfo.GetMinPage())
#0128            nStartPage = printInfo.GetMinPage();
#0129         if (nStartPage > printInfo.GetMaxPage())
#0130            nStartPage = printInfo.GetMaxPage();
#0131
#0132         int nStep = (nEndPage >= nStartPage) ? 1 : -1;
#0133         nEndPage = (nEndPage == 0xffff) ? 0xffff : nEndPage + nStep;
#0134
#0135         VERIFY(strTemp.LoadString(AFX_IDS_PRINTPAGENUM));
#0136
#0137         // begin page printing loop
#0138         BOOL bError = FALSE;
#0139  ①      for (printInfo.m_nCurPage = nStartPage;
#0140              printInfo.m_nCurPage != nEndPage; printInfo.m_nCurPage += nStep)
#0141         {
#0142  ②         OnPrepareDC(&dcPrint, &printInfo);
#0143
#0144            // check for end of print
#0145            if (!printInfo.m_bContinuePrinting)
#0146               break;
#0147
#0148            // write current page
#0149            TCHAR szBuf[80];
#0150            wsprintf(szBuf, strTemp, printInfo.m_nCurPage);
#0151  ③         dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_PAGENUM, szBuf);
#0152
#0153            // set up drawing rect to entire page (in logical coordinates)
#0154            printInfo.m_rectDraw.SetRect(0, 0,
#0155                    dcPrint.GetDeviceCaps(HORZRES),
#0156                    dcPrint.GetDeviceCaps(VERTRES));
#0157            dcPrint.DPtoLP(&printInfo.m_rectDraw);
#0158
#0159            // attempt to start the current page
#0160  ④         if (dcPrint.StartPage() < 0)
#0161            {
```

```
#0162                bError = TRUE;
#0163                break;
#0164             }
#0165
#0166             // must call OnPrepareDC on newer versions of Windows because
#0167             // StartPage now resets the device attributes.
#0168             if (afxData.bMarked4)
#0169                OnPrepareDC(&dcPrint, &printInfo);
#0170
#0171             ASSERT(printInfo.m_bContinuePrinting);

#0172
#0173             // page successfully started, so now render the page
#0174 ⑤          OnPrint(&dcPrint, &printInfo);
#0175 ⑥          if (dcPrint.EndPage() < 0 || !_AfxAbortProc(dcPrint.m_hDC, 0))
#0176             {
#0177                   bError = TRUE;
#0178                   break;
#0179             }
#0180          }
#0181
#0182          // cleanup document printing process
#0183          if (!bError)
#0184 ⑦          dcPrint.EndDoc();
#0185          else
#0186             dcPrint.AbortDoc();
#0187
#0188          AfxGetMainWnd()->EnableWindow();    // enable main window
#0189
#0190 ⑧       OnEndPrinting(&dcPrint, &printInfo);    // clean up after printing
#0191 ⑨       dlgPrintStatus.DestroyWindow();
#0192
#0193 ⑩       dcPrint.Detach();   // will be cleaned up by CPrintInfo destructor
#0194       }
#0195 }
```

图12-4 CView::OnFilePrint 原始代码，这是打印命令的第一战场。

标出号代码的是重要动作，稍后将有补充说明。

以下是CView::OnFilePrint函数之中重要动作的说明。你可以将这份说明与上一节「列印动作的背景原理」做一比对，就能够明白MFC在什么地方为我们做了什么事情，也才因此能够体会，究竟我们该在什么地方改写虚函数，放入我们自己的补强程序代码。

①OnFilePrint首先在堆栈中产生一个CPrintInfo对象，并构造之，使其部份成员变量拥有初值。CPrintInfo 是一个用来记录打印机数据的结构，其构造函数配置了一个Win32 通用打印对话框（common print dialog）并将它指定给 m_pPD：

```
// in AFXEXT.H
struct CPrintInfo // Printing information structure
{
    CPrintDialog* m_pPD; // pointer to print dialog
    BOOL m_bPreview; // TRUE if in preview mode
    BOOL m_bDirect; // TRUE if bypassing Print Dialog
    ...
};
```

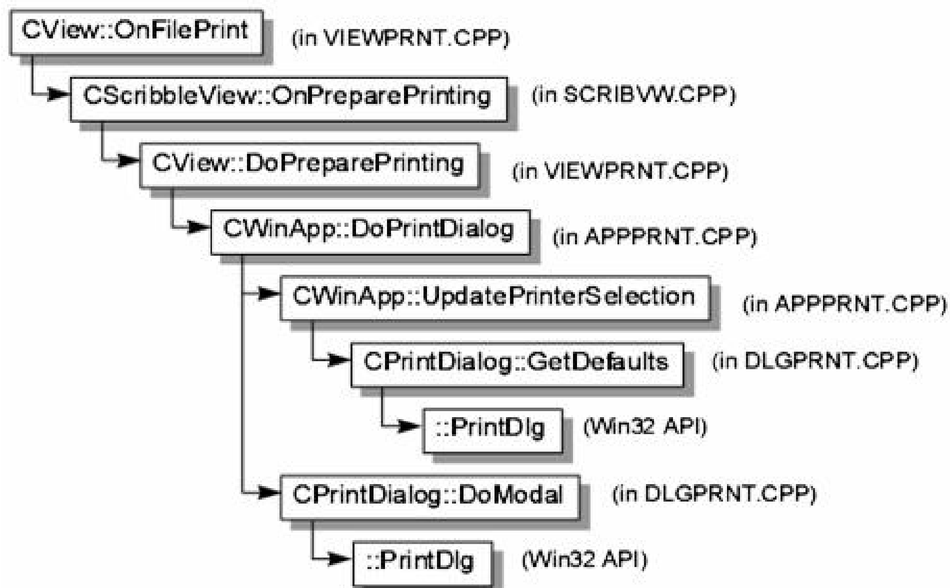
上述的成员变量m_bPreview 如果是TRUE，表示处于预览模式，FALSE 表示处于打印模式；成员变量m_bDirect 如果是TRUE，表示省略【打印】对话框，FALSE 表示需显示【打印】对话框。

上面出现过的CPrintDialog，用来更贴近描述打印对话框：

```
class CPrintDialog : public CCommonDialog
{
public:
    PRINTDLG& m_pd;
    BOOL GetDefaults();
    LPDEVMODE GetDevMode() const;   // return DEVMODE
    CString GetDriverName() const;  // return driver name
    CString GetDeviceName() const;  // return device name
    CString GetPortName() const; // return output port name
    HDC GetPrinterDC() const; // return HDC (caller must delete)
    HDC CreatePrinterDC();
    ...
};
```

②如果必要（从命令列参数中得知要直接打印某个文件到打印机上），利用::CreateDC产生一个「打印机DC」，并做打印动作。注意，printInfo.m_bDirect 被设为TRUE，表示跳过打印对话框，直接打印。

③OnPreparePrinting是一个虚函数，所以如果CView的派生类改写了它，控制权就移转到派生类手中。本例将移转到CScribbleView手中。CScribbleView::OnPreparePrinting 的预设内容（AppWizard自动为我们产生）是调用DoPreparePrinting，它并不是虚函数，而是CView的一个辅助函数。以下是其调用堆迭，直至【打印】对话框出现为止。



CView::DoPreparePrinting 将贮存在CPrintInfo结构中的对话框 CPrintDialog* m_pPD显示出来，借此收集使用者对打印机的各种设定，然后产生一个「打印机 DC」，储存在printinfo.m_pPD->m_pd.hDC 之中。

④如果使用者在【打印】对话框中选按【打印到文件】，则再显示一个【Print to File】对话框，让使用者设定文件名。



⑤接下来取文件名称和输出设备的名称（可能是打印机也可能是个文件），并产生一个DOCINFO结构，设定其中的 lpszDocName 和lpszOutput字段。此一 DOCINFO 结构将在稍后的StartDoc 动作中用到。

⑥如果使用者在【打印】对话框中按下【确定】钮，OnFilePrint 就在堆栈中制造出一个CDC对象，并把前面所完成的「打印机 DC」附着到CDC对象上：

```
CDC dcPrint;
dcPrint.Attach(printInfo.m_pPD->m_pd.hDC);
dcPrint.m_bPrinting = TRUE;
```

⑦一旦CDC完成，OnFilePrint把CDC对象以及前面的CPrintInfo 对象传给OnBeginPrinting 作为参数。OnBeginPrinting 是CView的一个虚函数，原本什么也没做。你可以改写它，设定打印前的任何初始状态。

⑧设定AbortProc。这应该是一个callback 函数，MFC有一个预设的简易函数_AfxAbortProc可兹利用。

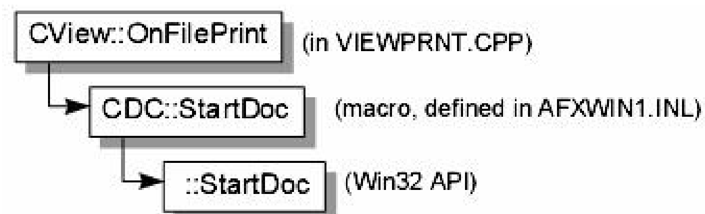


⑨把父窗口除能，产生【打印状态】对话框，根据文件名称以及输出设备名称，设定对话框内容，并显示之：

```
AfxGetMainWnd()->EnableWindow(FALSE);
CPrintingDialog dlgPrintStatus(this);
... // 设定对话框内容
dlgPrintStatus.ShowWindow(SW_SHOW);
dlgPrintStatus.UpdateWindow();
```



❿StartDoc通知打印机开始崭新的打印工作。这个函数其实就是启动Windows 打印引擎。

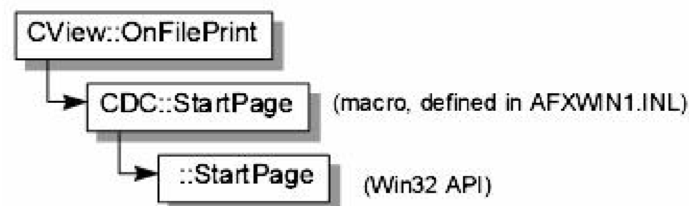


① 以 for循环针对文件中的每一页开始做打印动作。

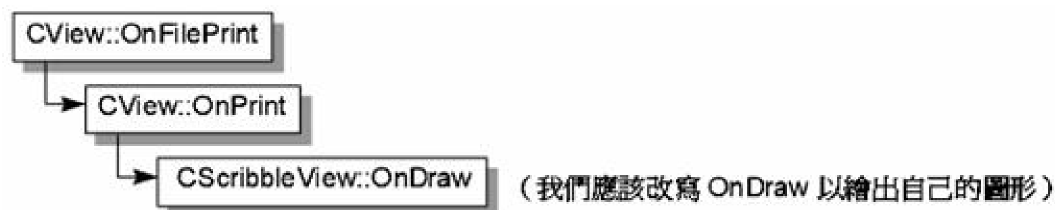
② 调用CView::OnPrepareDC。此函数什么也没做。如果你要在每页前面加表头，就请改写这个虚函数。

③ 修改【打印状态】对话框中的页次。

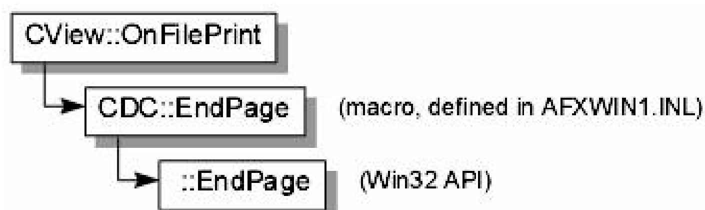
④ StartPage开始新的一页。



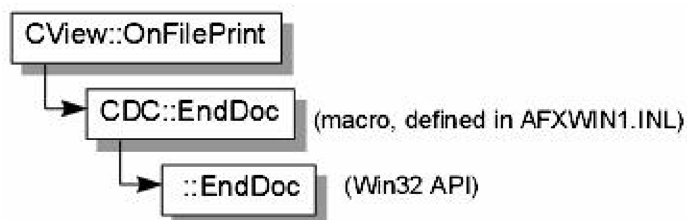
⑤ 调用CView::OnPrint，它的内部只有一个动作：调用OnDraw。我们应该在CScribbleView中改写OnDraw以绘出自己的图形。



⑥ 一页结束，调用dcPrint.EndPage



⑦ 文件结束，调用 EndDoc



⑧ 整个打印工作结束。如果有些什么绘图资源需要释放，你应该改写 OnEndPrinting函数并在其中释放之。



⑨ 去除【打印状态】对话框。

⑩ 将「打印机 DC」解除附着，CPrintInfo 的析构函数会把DC还给Windows。从上面这些分析中归纳出来的结论是，一共有六个虚函数可以改写，请看图 12-5。

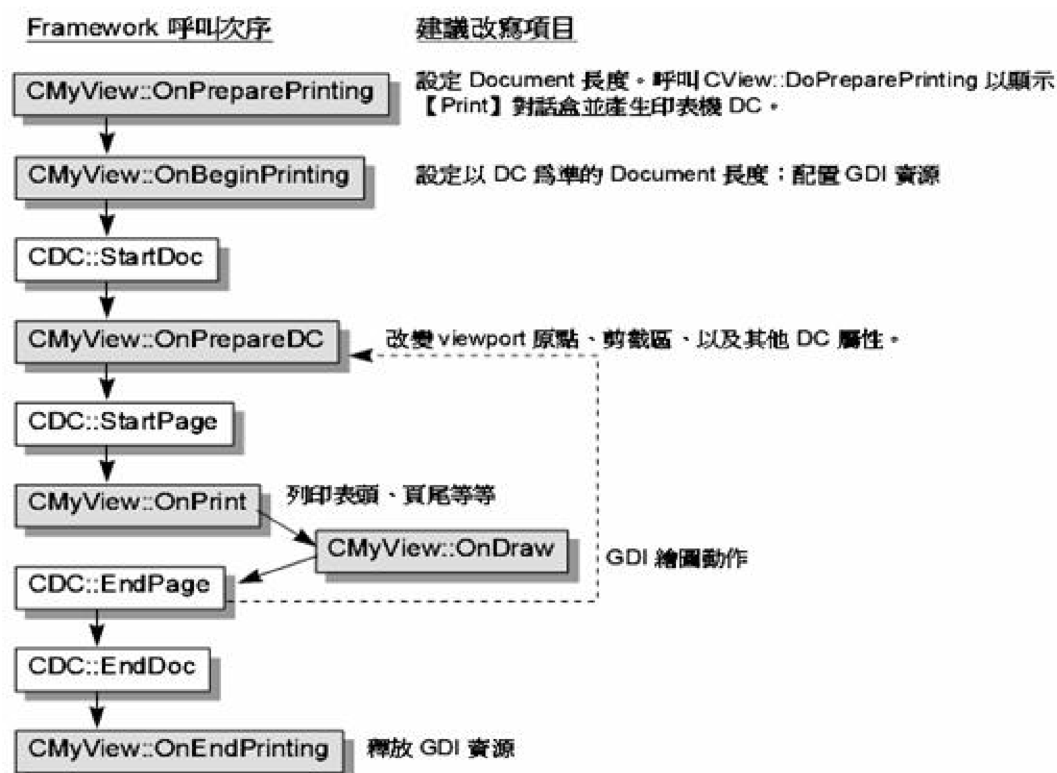


图12-5 MFC打印流程与我们的着力点

以下是图12-5的补充说明。

- 当使用者按下【File/Print】命令项， Application Framework首先调用CMyView::OnPreparePrinting。这个函数接受一个CPrintInfo 指针做为参数，允许使用者设定Document 的打印长度（从第几页到第几页）。预设页代码是1至0xFFFF， 程序员

应该在OnPreparePrinting中 呼 叫SetMaxPage 预 设 页数 。SetMaxPage 之后，程序应该调用CView::DoPreparePrinting，它会显示【打印】对话框，并产生一个打印机DC。当对话框结束，CPrintInfo 也从中获得了使用者设定的各个印表项目（例如从第n1页印到第n2页)。

Framework 如何得知使用者对于打印状态的设定？CPrintInfo 有五个函数可用，下一节有更详细的说明。

- 针对每一页，Framework 会调用CMyView::OnPrepareDC，这函数在前一章介绍CScrollView 时也曾提过，当时是因为我们使用卷动窗口，而由于卷动的关系，绘图之前必须先设定DC的映射模式和原点等性质。这次稍有不同的是，它收到打印机DC做为第一参数，CPrintInfo 对象做为第二参数。我们改写这个函数，使它依目前的页代码来调整DC，例如改变打印原点和截割局部以保证印出来的Document内容的合适性等等。
- 稍早我一再强调所有绘图动作都应该集中在OnDraw 函数中，Framework 会自动调用它。更精确地说，Framework 其实是先调用OnPrint，传两个参数进去，第一参数是个DC，第二参数是个CPrintInfo指针。OnPrint内部再调用OnDraw，这次只传DC过去，做为唯一参数：

```
// in VIEWCORE.CPP
void CView::OnPrint(CDC* pDC, CPrintInfo*)
{
    ASSERT_VALID(pDC);
    // Override and set printing variables based on page number
    OnDraw(pDC);     // Call Draw
}
```

有了这样的差异，我们可以这么区分这两个函数的功能：

- OnPrint：负 责「只在印表时才做（屏幕显示时不做）」的动作。例如印出

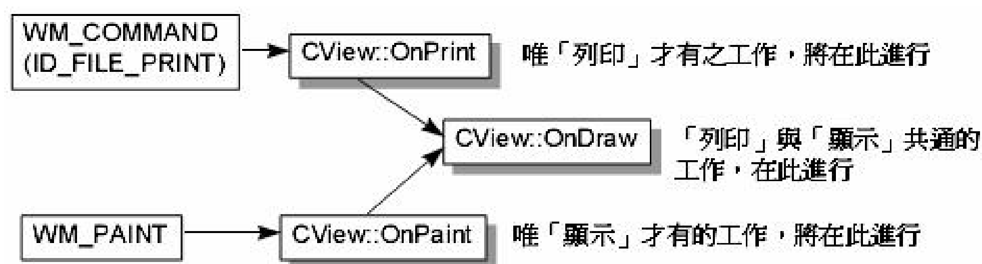
表头和页尾。

- OnDraw ：共通性绘图动作（包括输出到屏幕或打印机上）都在此完成。

看看另一个函数OnPaint：

```
// in VIEWCORE.CPP
void CView::OnPaint()
{
    // standard paint routine
    CPaintDC dc(this);
    OnPrepareDC(&dc);
    OnDraw(&dc);
}
```

你会发现原来它们是这么分工的：



所谓「显示」是指输出到屏幕上，「打印」是指输出到打印机上。

由同一函数完成显示（display）与打印（print）动作，才能够达到「所见即所得」（What You See Is What You Get，WYSIWYG）的目的。如果你不需要一个 WYSIWYG 程序，可以改写OnPrint使它不要调用OnDraw，而调用另一个绘图例程。

不要认为什么情况下都需要WYSIWYG。一个文字编辑器可能使用粗体字打印但使用控制代码在屏幕上代表这粗体字。

Scribble 打印机制的补强

MFC 预设的打印机制够聪敏了，但还没有聪敏到解决所有的问题。这些问题包括：

- 打印出来的影像可能不是你要的大小
- 不会分页
- 没有表头（header）
- 没有页尾（footer）

毕竟屏幕输出和打印机输出到底还是有着重大的差异。窗口有卷动杆而打印机没有，这伴随而来的就是必须计算Document的大小和纸张的大小，以解决分页的问题；此外，我们必须想想，在MFC预设的打印机制中，改写哪一个地方，才能让我们有办法在Document 的输出页加上表头或页尾。

打印机的页和文件的页

首先，我们必须区分「页」对于 Document 和对于打印机的不同意义。从打印机观点来看，一页就是一张纸，然而一张纸并不一定容纳 Document 的一页。例如你想印一些通讯数据，这些数据可能是要被折迭起来的，因此一张纸印的是 Document 的第一页和最后一页（亲爱的朋友，想想你每天看的报纸）。又例如印一个巨大的电子表格，它可能是Document 上的一页，却占据两张 A4 纸。

MFC这个Application Framework 把关于打印的大部份信息都记录在 CPrintInfo 中，其中数笔数据与分页有密切关系。下表是取得分页数据的相关成员，其中只有SetMaxPage和m_nCurPage 和m_nNumPreviewPages在 Scribble 程序中会用到，原因是Scribble程序对许

多问题做了简化。

```
CPrintInfo成员名称      参考到的打印页
GetMinPage/SetMinPage   Document 中的第一页
GetMaxPage/SetMaxPage   Document 中的最后一页
GetFromPage           将被印出的第一页（出现在【打印】对话框，图12-1b）
GetToPage             将被印出的最后一页（出现在【打印】对话框）
m_nCurPage            目前正被印出的一页（出现在【打印状态】对话框）
m_nNumPreviewPages        预览窗口中的页数（稍后将讨论之）
```

注：页代码从 1（而不是 0）开始。

CPrintInfo 结构中记录的「页」数，指的是打印机的页数；Framework 针对每一「页」调用 OnPrepareDC以及OnPrint 时，所指的「页」也是打印机的页。当你改写OnPreparePrinting 时指定Document的长度，所用的单位也是打印机的「页」。如果Document的一页恰等于打印机的一页（一张纸），事情就单纯了；如果不是，你必须在两者之间做转换。

Scribble Step5 设定让每一份 Document 使用打印机的两页。第一页只是单纯印出文件名称（文件名称），第二页才是文件内容。假设我利用 View 窗口卷动杆在整个Document四周画一四方圈的话，我希望这一四方圈落入第二页（第二张纸）中。当然，边界留白必须考虑在内，如图 12-6。除此之外，我希望第二页（文件内容）最顶端留一点空间，做为表头。本例在表头中放的是文件名称。

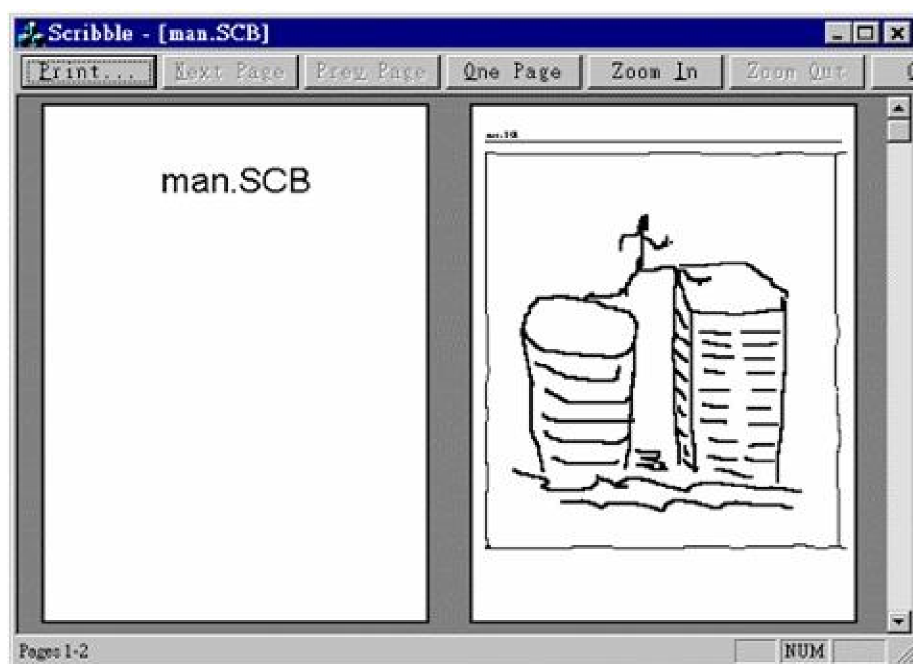


图12-6 Scribble Step5 的每一份文件打印时有两页，

第一页是文件名称，第二页是文件内容，最顶端留有一个表头。

配置 GDI 绘图工具

绘图难免需要各式各样的笔、刷、颜色、字形、工具。这些 GDI 资源都会占用内存，而且是GDI模块的heap。虽说Windows 95 对于USER 模块和GDI模块的heap已有大幅改善，使用32 位 heap，不再局限 64KB，但我们当然仍然不希望看到浪费的情况发生，因此最好的方式就是在打印之前配置这些 GDI 绘图对象，并在打印后立刻释放。

看看图 12-5，配置GDI对象的最理想时机显然是OnBeginPrinting，两个理由：

1. 每当Framework 开始一份新的打印工作，它就会调用此函数一次，因此不同打印工作所需的不同工具可在此有个替换。

2. 此函数的参数是一个和「打印机DC」有附着关系的CDC 对象指针，我们直接从此一CDC 对象中配置绘图工具即可。

配置得来的GDI对象可以储存在View的成员变量中，供整个打印过程使用。使用时机当然是OnPrint。如果你必须对不同的打印页使用不同的GDI对象，CPrintInfo中的m_nCurPage 可以帮你做出正确的决定。

释放GDI对象的最理想时机当然是在OnEndPrinting，这是每当一份打印工作结束后，Application Framework 会调用的函数。

Scribble 没有使用什么特殊的绘图工具，因此下面这两个虚函数也就没有修改，完全保留AppWizard当初给我们的样子：

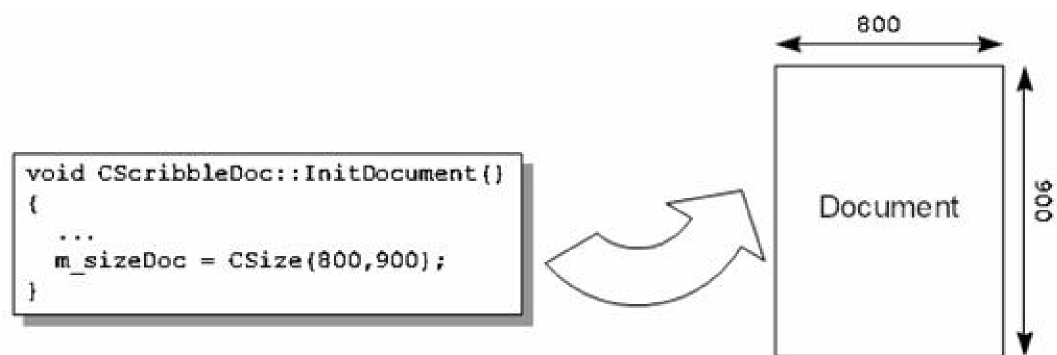
```
void CScribbleView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
       // TODO: add extra initialization before printing
}

void CScribbleView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
       // TODO: add cleanup after printing
}
```

尺寸与方向：关于映射模式（坐标系统）

回忆所谓的坐标系统，我已经在上一章描述过CScrollView如何为了卷动效果而改变座标系统的原点。除了改变原点，我们甚至可以改变坐标系统的单位长度，乃至于改变座标系统的横纵比例（scale）。这些就是这一节要讨论的重点。

Document 有大小可言吗？有的，在打印过程中，为了计算Document 对应到打印机的页数，我们需要Document 的尺寸。CScribbleDoc 的成员变量 m_sizeDoc，就是用来记录Document 的大小。它是一个CSize 对象：



事实上，所谓「逻辑坐标」原本是没有大小的，如果我们说一份Document宽 800 高900，那么若逻辑坐标的单位是英吋，这就是8英吋宽9 英吋高；若逻辑坐标的单位是公分，这就是8公分宽9 公分高。如果逻辑单位是图素（Pixel）呢？那就是 800 个图素宽 900 个图素高。图素的大小随着输出装置而改变，在 14 吋 Super VGA（1024x768）显示器上，800x900 个图素大约是 21.1 公分宽 23.6 公分高，而在一部 300 DPI（Dot Per Inch，每英吋点数）的激光打印机上，将是2-2/3 英吋宽3英吋高。

预设情况下GDI绘图函数使用MM_TEXT 映射模式（Mapping Mode，也就是坐标系统，注），于是逻辑坐标等于装置坐标，也就是说一个逻辑单位是一个图素。如果不重新设定映射模式，可以想见屏幕上的图形一放到300 DPI 打印机上都嫌太小。

解决的方法很简单：设定一种与真实世界相符的逻辑坐标系统。Windows 提供的八种映像模式中有七种是所谓的metric 映射模式，它们的逻辑单位都建立在公分或英吋的基础上，这正是我们所要的。如果把 OnDraw 内的绘图动作都设定在 MM_LOENGLISH 映射模式上（每单位 0.01 英吋），那么不论输出到屏幕上或到打印机上都获得相同的尺度。真正要为「多少图点才能画出一英吋长」伤脑筋的是装置驱动程序，不是我们。

注：GDI 的八种映射模式及其意义如下：

- MM_TEXT：以图素（pixel）为单位，Y 轴向下为正，X 轴向右为正。
- MM_LOMETRIC：以0.1 公分为单位，Y 轴向上为正，X 轴向右为正。
- MM_HIMETRIC：以0.01 公分为单位，Y 轴向上为正，X 轴向右为正。
- MM_LOENGLISH：以0.01 英吋为单位，Y 轴向上为正，X 轴向右为正。
- MM_HIENGLISH：以0.001 英吋为单位，Y 轴向上为正，X 轴向右为正。
- MM_TWIPS：以1/1440 英吋为单位，Y 轴向上为正，X 轴向右为正。
- MM_ISOTROPIC：单位长度可任意设定，Y 轴向上为正，X 轴向右为正。
- MM_ANISOTROPIC：单位长度可任意设定，且X 轴单位长可以不同于Y 轴单位长（因此圆可能变形）。Y 轴向上为正，X 轴向右为正。

回忆上一章为了卷动窗口，曾有这样的动作：

```
void CScribbleView::OnInitialUpdate()
{
    SetScrollSizes(MM_TEXT, GetDocument()->GetDocSize());
    CScrollView::OnInitialUpdate();
}
```

映射模式可以在 SetScrollSizes 的第一个参数指定。现在我们把它改为：

```
void CScribbleView::OnInitialUpdate()
{
    SetScrollSizes(MM_LOENGLISH, GetDocument()->GetDocSize());
    CScrollView::OnInitialUpdate();
}
```

注意，OnInitialUpdate 更在OnDraw之前被调用，也就是说我们在真正绘图动作OnDraw之前完成了映射模式的设定。

映射模式不仅影响逻辑单位的尺寸，也影响Y轴坐标方向。MM_TEXT是Y轴向下，MM_LOENGLISH（以及其它任何映射模式）是 Y 轴向上。但，虽然有此差异，我们的Step5程序代码却不需为此再做更动，因为DPtoLP已经完成了这个转换。别忘了，鼠标左键传来的点坐标是先经过DPtoLP 才储存到CStroke对象并且然后才由LineTo 画出的。

然而，程序的某些部份还是受到了Y轴方向改变的冲击。映射模式只会改变 GDI各相关函数，不使用DC的地方，就不受映射模式的影响，例如CRect的成员函数就不知晓所谓的映射模式。于是，本例中凡使用到 CRect 的地方，要特别注意做些调整：

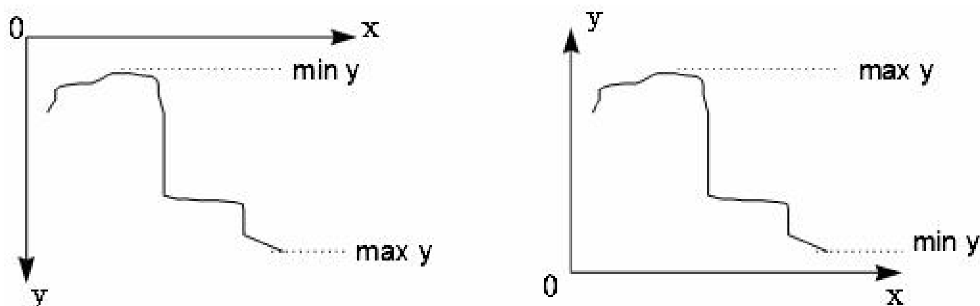
1. 修正「线条外围四方形」的计算方式。原计算方式是在 FinishStroke 中这么做：

```
for (int i=1; i < m_pointArray.GetSize(); i++)
{
    pt = m_pointArray[i];
    m_rectBounding.left   = min(m_rectBounding.left, pt.x);
    m_rectBounding.right  = max(m_rectBounding.right, pt.x);
    m_rectBounding.top    = min(m_rectBounding.top, pt.y);
    m_rectBounding.bottom = max(m_rectBounding.bottom, pt.y);
}
m_rectBounding.InflateRect(CSize(m_nPenWidth, m_nPenWidth));
```

新的计算方式是：

```
for (int i=1; i < m_pointArray.GetSize(); i++)
{
    pt = m_pointArray[i];
    m_rectBounding.left = min(m_rectBounding.left, pt.x);
    m_rectBounding.right = max(m_rectBounding.right, pt.x);
    m_rectBounding.top = max(m_rectBounding.top, pt.y);
    m_rectBounding.bottom = min(m_rectBounding.bottom, pt.y);
}
m_rectBounding.InflateRect(CSize(m_nPenWidth, -(int)m_nPenWidth));
```

这是因为在 Y 轴向下的系统中，四方形的最顶点位置应该是找Y坐标最小者；而在Y 轴向上的系统中，四方形的最顶点位置应该是找Y坐标最大者；同理，对于四方形的最底点亦然。



2. 我们在OnDraw中曾经以IntersectRect计算两个四方形是否有交集。这个函数也是CRect成员函数，它假设：一个四方形的底坐标 Y 值必然大于顶坐标的Y 值（这是从装置坐标，也就是MM_TEXT，的眼光来看)；如果事非如此，它根本不可能找出两个四方形的交集。因此我们必须在 OnDraw 中做以下修改，把逻辑坐标改为装置坐标：

```
void CScribbleView::OnDraw(CDC* pDC)
{
        CScribbleDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        // Get the invalidated rectangle of the view, or in the case
        // of printing, the clipping region of the printer dc.
        CRect rectClip;
        CRect rectStroke;
        pDC->GetClipBox(&rectClip);
        pDC->LPtoDP(&rectClip);
        rectClip.InflateRect(1, 1); // avoid rounding to nothing

        // Note: CScrollView::OnPaint() will have already adjusted the
        // viewport origin before calling OnDraw(), to reflect the
        // currently scrolled position.

        // The view delegates the drawing of individual strokes to
        // CStroke::DrawStroke().
        CTypedPtrList<CObList,CStroke*>& strokeList = pDoc->m_strokeList;
        POSITION pos = strokeList.GetHeadPosition();
        while (pos != NULL)
        {
                CStroke* pStroke = strokeList.GetNext(pos);
                rectStroke = pStroke->GetBoundingRect();

                pDC->LPtoDP(&rectStroke);
                rectStroke.InflateRect(1, 1); // avoid rounding to nothing
                if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
                        continue;
                pStroke->DrawStroke(pDC);
        }
}
```

分页

Scribble 程序的Document 大小固定是800x900，而且我们让它填满打印机的一页。因此 Scribble 并没有「将 Document分段打印」这种困扰。如果真要分段打印，Scribble 应该改写 OnPrepareDC，在其中视打印的页数调整 DC 的原点和截割局部。

即便如此，Scribble 还是在分页方面加了一些动作。本例一份 Document 打印时被视为一张标题和一张图片的组合，因此打印一份 Document 固定要耗掉两张印表纸。我们可以这么设计：

```
BOOL CScribbleView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMaxPage(2); // 文件总共有两页经线：
    // 第一页是标题页 (title page)
    // 第二页是文件页 (图形)
    BOOL bRet = DoPreparePrinting(pInfo); // default preparation
    pInfo->m_nNumPreviewPages = 2;  // Preview 2 pages at a time
    // Set this value after calling DoPreparePrinting to override
    // value read from .INI file
    return bRet;
}
```

接下来打算设计一个函数用以输出标题页，一个函数用以输出文件页。后者当然应该由 OnDraw 负责啰，但因为这文件页不是单纯的 Document 内容，还有所谓的表头，而这是打印时才做的东西，屏幕显示时并不需要的，所以我们希望把列印表头的工作独立于OnDraw 之外，那么最好的安置地点就是 OnPrint 了（请参考图12-5之后的补充说明的最后一点）。

Scribble Step5 把列印表头的工作独立为一个函数。总共这三个额外的函数应该声明于 SCRIBBLEVIEW.H 中，其中的PrintPageHeader 在下一节列出。

```
class CScribbleView : public CScrollView
{
public:
        virtual void OnPrint(CDC* pDC, CPrintInfo* pInfo);
        void PrintTitlePage(CDC* pDC, CPrintInfo* pInfo);
        void PrintPageHeader(CDC* pDC, CPrintInfo* pInfo, CString& strHeader);
  ...
}

#0001  void CScribbleView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
#0002  {
#0003      if (pInfo->m_nCurPage == 1)  // page no. 1 is the title page
#0004      {
#0005              PrintTitlePage(pDC, pInfo);
#0006              return; // nothing else to print on page 1 but the page title
#0007      }
#0008      CString strHeader = GetDocument()->GetTitle();
#0009
#0010      PrintPageHeader(pDC, pInfo, strHeader);
#0011      // PrintPageHeader() subtracts out from the pInfo->m_rectDraw the
#0012      // amount of the page used for the header.
#0013
#0014      pDC->SetWindowOrg(pInfo->m_rectDraw.left,-pInfo->m_rectDraw.top);
#0015
#0016      // Now print the rest of the page
#0017      OnDraw(pDC);
#0018  }
#0019
#0020  void CScribbleView::PrintTitlePage(CDC* pDC, CPrintInfo* pInfo)
#0021  {
#0022      // Prepare a font size for displaying the file name
#0023      LOGFONT logFont;
#0024      memset(&logFont, 0, sizeof(LOGFONT));
#0025      logFont.lfHeight = 75;  //  3/4th inch high in MM_LOENGLISH
#0026                                                 // (1/100th inch)

#0027      CFont font;
#0028      CFont* pOldFont = NULL;
#0029      if (font.CreateFontIndirect(&logFont))
#0030              pOldFont = pDC->SelectObject(&font);
#0031
#0032      // Get the file name, to be displayed on title page
#0033      CString strPageTitle = GetDocument()->GetTitle();

#0034
#0035      // Display the file name 1 inch below top of the page,
#0036      // centered horizontally
#0037      pDC->SetTextAlign(TA_CENTER);
#0038      pDC->TextOut(pInfo->m_rectDraw.right/2, -100, strPageTitle);
#0039
#0040      if (pOldFont != NULL)
#0041              pDC->SelectObject(pOldFont);
#0042  }
```

表头与页尾

文件名称以及文件内容的页代码应该有地方呈现出来。屏幕上没有问题，文件名称可以出现在窗口标题，页代码可以出现在状态列；但输出到打印机上时，我们就应该设计文件的表头与页尾，分别用来放置文件名称与页代码，或其它任何你想要放的数据。显然，即使是「所见即所得」，在打印机输出与屏幕输出两方面仍然存在至少这样的差异。

我们设计了另一个辅助函数，专门负责列印表头，并将OnPrint 的参数（一个打印机 DC）传给它。有一点很容易被忽略，那就是你必得在 OnPrint 调用 OnDraw 之前调整窗口的原点和范围，以避免该页的主内容把表头页尾给盖掉了。

要补偿被表头页尾占据的空间，可以利用CPrintInfo结构中的 m_rectDraw，这个字段记录着本页的可绘图局部。我们可以在输出主内容之前先输出表头页尾，然后扣除m_rectDraw 四方形的一部份，代表表头页尾所占空间。OnPrint 也可以根据m_rectDraw的数值决定有多少内容要放在打印页的主体上。

我们甚至可能因为表头页尾的加入，而需要修改 OnDraw，因为能够放到一张印表纸上的文件内容势必将因为表头页尾的出现而减少。不过，还好本例并不是这个样子。本例不设页尾，而文件大小在 MM_LOENGLISH 映射模式下是8英吋宽9 英吋高，放在一页A4 纸张（210 x 297 毫米）或Letter Size（8-1/2 x 11英吋）纸张中都绰绰有余。

```
#0001 void CScribbleView::PrintPageHeader(CDC* pDC, CPrintInfo* pInfo,
#0002         CString& strHeader)
#0003 {
#0004     // Print a page header consisting of the name of
#0005     // the document and a horizontal line
#0006     pDC->SetTextAlign(TA_LEFT);
#0007     pDC->TextOut(0,-25, strHeader);  // 1/4 inch down
#0008
#0009     // Draw a line across the page, below the header
#0010     TEXTMETRIC textMetric;
#0011     pDC->GetTextMetrics(&textMetric);
#0012     int y = -35 - textMetric.tmHeight;    // line 1/10th inch below text
#0013     pDC->MoveTo(0, y);                     // from left margin
#0014     pDC->LineTo(pInfo->m_rectDraw.right, y);  // to right margin
#0015
#0016     // Subtract out from the drawing rectange the space used by the header.
#0017     y -= 25;    // space 1/4 inch below (top of) line
#0018     pInfo->m_rectDraw.top += y;
#0019 }
```

动态计算页代码

某些情况下View类在开始打印之前没办法事先知道Document的长度。假设你的程序并不支持「所见即所得」，那么屏幕上的 Document 就不会对应到它打印时真正的长度。这就引起了一个问题，你没有办法在改写 OnPreparePrinting 时，利用SetMaxPage 为 CPrintInfo 结构设定一个最大页代码，因为这时候的你根本不知道 Document的长度。而如果使用者不能够在【打印】对话框中指定「结束页代码」，Framework 也就不知道何时才停止打印的循环。唯一的方法就是边印边看，View 类必须检查是否目前已经印到Document 的尾端，并在确定之后通知Framework。

那么我们的当务之急是找出在哪一个点上检查Document结束与否，以及如何通知Framework停止打印。从图 12-5 可知，打印的循环动作的第一个函数是 OnPrepareDC，我们可以改写此一函数，在此设一道关卡，如果检查出 Document 已到尾端，就要求中止打印。

Framework是否结束打印，其实全赖CPrintInfo 的m_bContinuePrinting 字段。此字段如果是FALSE，Framework 就中止打印。预设情况下OnPrepareDC 把此字段设为FALSE。小心，这表示如果Document 长度没有指明，Framework 就假设这份Document只有一页长。因此你在调用基类的OnPrepareDC 时需格外注意，可别总以为m_bContinuePrinting 是TRUE。

打印预览（**Print Preview**）

什么是打印预览？简单地说，把屏幕仿真为打印机，将图形输出于其上就是了。预览的目的是为了让使用者在打印机输出之前，先检查他即将获得的成果，检查的重要项目包括图案的布局以及分页是否合意。

为了完成预览功能，MFC 在CDC之下设计了一个子类，名为CPreviewDC。所有其他的CDC对象都拥有两个DC，它们通常井水不犯河水；然而 CPreviewDC 就不同，它的第一个DC表示被仿真的打印机，第二个DC是真正的输出目的地，也就是屏幕（预览结果输出到屏幕，不是吗 ?!）

一旦你选择【File/Print Preview】命令项，Framework 就产生一个 CPreviewDC 对象。只要你的程序曾经设定打印机DC的特征（即使没有动手设定，也有其默认值），Framework就会把同样的性质也设定到 Preview DC 上。举个例子，你的程序选择了某种打印字形，Framework 也会对屏幕选择一个仿真打印机输出的字形。一旦程序要做打印预览，Framework 就透过仿真的打印机 DC，再把结果送到显示屏DC去。

为什么我不再像前面那样去看整个预览过程中的调用堆栈并追踪其原始代码呢？因为预览对我们而言太完善了，几乎不必改写什么虚函数。唯一在 Scribble Step5 中与打印预览有关系的，就是下面这一行：

```
BOOL CScribbleView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMaxPage(2); // the document is two pages long:
    // the first page is the title page
    // the second is the drawing
    BOOL bRet = DoPreparePrinting(pInfo); // default preparation
    pInfo->m_nNumPreviewPages = 2;  // Preview 2 pages at a time
    // Set this value after calling DoPreparePrinting to override
    // value read from .INI file
    return bRet;
}
```

现在，Scribble Step5 全部完成。

本章回顾

前面数章中早就有了打印功能，以及预览功能。我们什么也没做，只不过在 AppWizard 的第四个步骤中选了【Printing and Print Preview】项目而已。这足可说明 MFC 为我们做掉了多少工作。想想看，一整个打印与预览系统耶。

然而我们还是要为打印付出写代码代价，原因是预设的打印大小不符理想，再者当我们想加点标题、表头、页尾时，必得亲自动手。

延续前面的风格，我还是把 MFC 提供的打印系统的背后整个原理挖了出来，使你能够清楚知道在哪里下药。在此之前，我也把 Windows 的打印原理（非关 MFC）整理出来，这样你才有循序渐进的感觉。然后，我以各个小节解释我们为 MFC 打印系统所做的补强工作。

现在的Scribble，具备了绘图能力，文件读写能力，打印能力，预览能力，丰富的窗口表现能力。除了Online Help 以及OLE 之外，所有大型软件该具备的能力都有了。我并不打算在本书之中讨论Online Help，如果你有兴趣，可以参考 Visual C++ Tutorial（可在Visual C++ 的 Online 数据中获得）第10 章。

我也不打算在本书之中讨论 OLE，那牵扯太多技术，不在本书的设定范围。

Scribble Step5 的完整原始代码，列于附录 B。

第 13 章 多重文件与多重显示

你可能会以【Window/New Window】为同一份文件制造出另一个 View 窗口，也可能设计分裂窗口，以多个窗口呈现文件的不同角落 （如第 11 章所为）。但，这两种情况都是以相同的显示方式表达文件的内容。

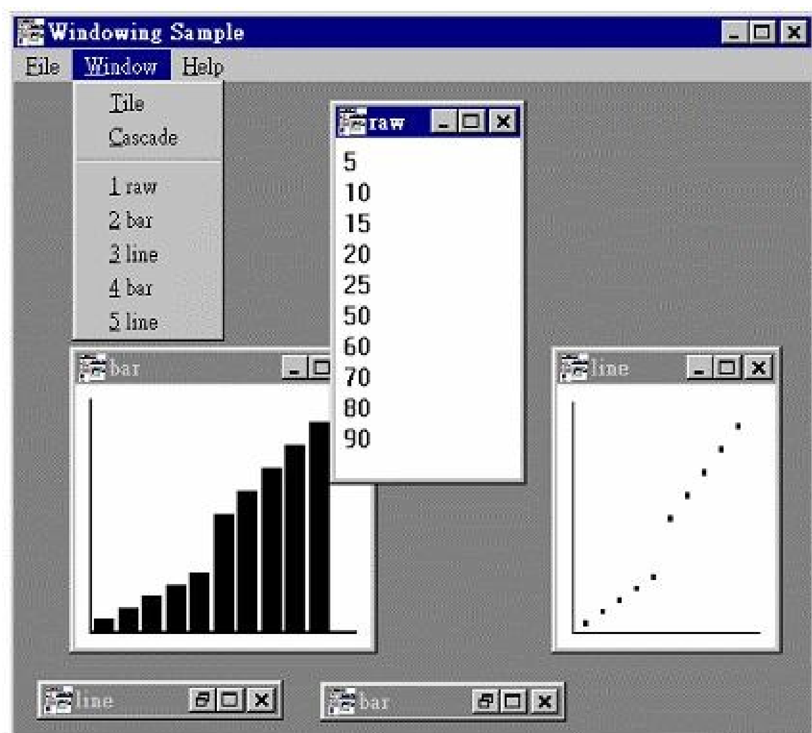
如何突破一成不变的显示方法，达到丰富的表现效果？

这一章我将对于 Document/View 再作各种深入应用。重要放在显象技术以及多重文件的技术上。

MDI和SDI

首先再让我把MDI和SDI的观念厘清楚。

在传统的SDK 程序设计中，所谓MDI是指「一个大外框窗口，内部可容纳许多小子窗口」的这种程序风格。内部的小子窗口即是「Document 窗口」-- 虽然当时并未有如MFC 所谓的Document 观念。此外，「MDI 风格」还包括程序必须有一个Window 选单，提供对于小子窗口的管理，包括tile、cascade、icon arrange 等命令项：



至于SDI程序，就是一般的、没有上述风格的non-MDI程序。

在MFC的定义中，MDI表示可「同时」开启一份以上的Documents，这些 Documents可以是相同类型，也可以是不同类型。许多份Documents 同时存在，必然需要许多个子窗口容纳之，每个子窗口其实是Document的一个View。即使你在 MDI 程序中只开启一份 Document,

但以【Window/New Window】的方式打开第二个view、第三个view...，亦需占用多个子窗口。因此这和 SDK 所定义的 MDI 有异曲同工的意义。

至于SDI 程序，同一时间只能开启一份Document。一份Document 只占用一个子窗口（亦即其 View 窗口），因此这也与 SDK所定义的SDI意义相同。当你要在SDI程序中开启第二份Document，必须先把第一份Document关闭。MDI 程序未必一定得提供一个以上的Document 类型。所谓不同的 Document 类型是指程序提供不同的CDocument 派生类，亦即有不同的Document Template。软件工业早期曾经流行一种「全效型」软件，既处理电子表格、又作文书处理、又能绘图作画，伟大得不得了，这种软件就需要数种文件类型：电子表格、文书、图形。

多重显像（Multiple Views）

只要是具备 MDI 性质的 MFC 程序（也就是你曾在 AppWizard 步骤一中选择【Multiple Documents】项目），天生就具备了「多重显像」能力。「天生」的意思是你不必动手，application framework 已经内含了这项功能：随便执行任何一版的 Scribble，你都可以在【Window】选单中找到【New Window】这个命令项，按下它，就可以获得「同源子窗口」如图 13-1。

我将以「多重显像」来称呼 Multiple Views。多重显像的意思是数据可以不同的类型显现出来。并以「同源子窗口」代表「显示同一份 Document 而又各自分离的View 窗口」。

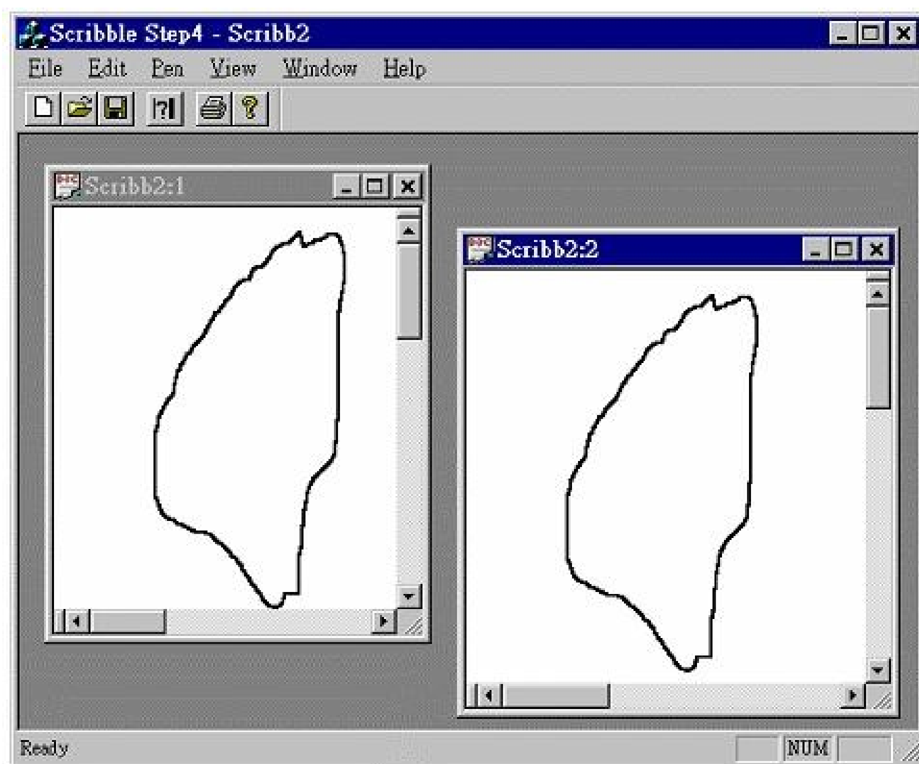


图13-1 【 Window/New Window】可以为「目前作用中的

View 所对应的Document 再开一个View 窗口。

另外，第11章也介绍了一种变化，是利用分裂窗口的各个窗口，显示 Document 内容。这些窗口虽然集中在一个大窗口中，但它们的视野却可以各自独立，也就是说它们可以看到 Document 中的不同局部，如图 13-2。

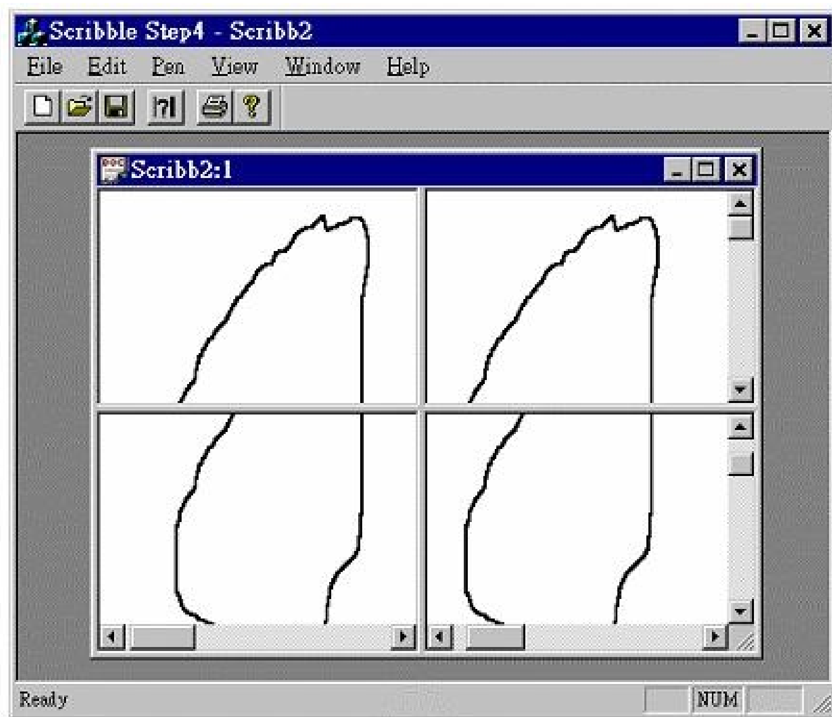


图13-2 分裂窗口的不同窗口可以观察同一Document数据的不同局部。

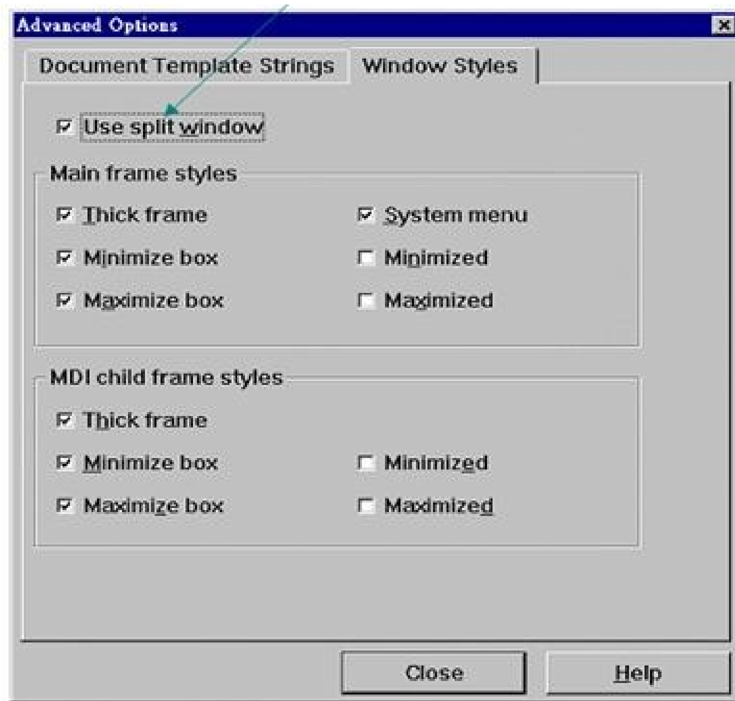
但是我们发现，不论是同源子窗口或分裂窗口的窗口，都是以相同的方式（也就是同一个CMyView::OnDraw）表现Document内容。如果我们希望表达力丰富一些，如何是好？到现在为止我们并没有看到任何一个Scribble 版本具备了多种显像能力。

窗口的动态分裂

动态分裂窗口由CSplitterWnd 提供服务。这项技术已经在第11章的 Scribble Step4 示范过了。它并没有多重显像的能力，因为每一个窗口所使用的 View 类完全相同。当第一个窗口形成（也就是分裂窗口初产生的时候），它将使用Document Template中登记的View类，作为其View类。尔后当分裂发生，也就是当使用者拖拉滚动条之上名为分裂棒（splitter box）的横杆，导至新窗口诞生，程序就以「动态生成」的方式产生出新的View窗口。

因此，View 类一定必须支持动态生成，也就是必须使用 DECLARE_DYNCREATE 和 IMPLEMENT_DYNCREATE 宏。请回顾第 8 章。

AppWizard 支持动态分裂窗口。当你在AppWizard 步骤四的【Advanced】对话框的【Windows Styles】附页中选按【Use split window】选项：



你的程序比起一般未选【Use split window】选项者有如下差异（阴影部份）：

```
// in CHILDFRM.H
class CChildFrame : public CMDIChildWnd
{
...
protected:
        CSplitterWnd m_wndSplitter;

public:
// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CChildFrame)
        public:
        virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext);
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        //}}AFX_VIRTUAL
...
};

// in CHILDFRM.CPP
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
        CCreateContext* pContext)
{
        return m_wndSplitter.Create( this,
                2, 2,           // TODO: adjust the number of rows, columns
                CSize( 10, 10 ),   // TODO: adjust the minimum pane size
                pContext );
}
```

✦ CSplitterWnd::Create 的详细规格请回顾第11章。

这些其实也就是我们在第 11 章为 Scribble Step4 亲手加上的代码。如果你一开始就打定主意要使用动态分裂窗口，如上便是了。

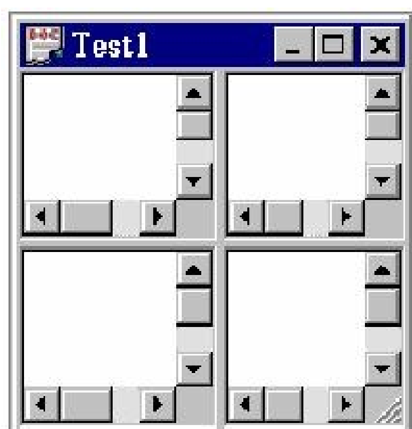
窗口（Panes）之间的同步更新，其机制着落在两个虚函数 CDocument::UpdateAllViews和 CView::OnUpdate 身上，与第 11 章的情况完全相同。

动态分裂的实现，非常简单。但它实在称不上「怎么样」！除了拥有「动态」增减窗口的长处之外，短处有二：第一，每一个窗口都使用相同的 View 类，因此显示出来的东西千篇一律；第二，窗口之间并非完全独立。同一水平列的窗口，使用同一个垂直卷轴；同一垂直行的窗口，使用同一个水平滚动条，如图 13-2。

窗口的静态分裂

动态分裂窗口的短处正是静态分裂窗口的长处， 动态分裂窗口的长处正是静态分裂窗口的短处。

静态分裂窗口的窗口个数一开始就固定了，窗口所使用的 view 必须在分裂窗口诞生之际就准备好。每一个窗口的活动完全独立自主，有完全属于自己的水平滚动条和垂直滚动条。



静态分裂窗口的窗口个数限制是 16 列 x 16 行，

动态分裂窗口的窗口个数限制是 2 列 x 2 行。

欲使用静态分裂窗口，最方便的办法就是先以 AppWizard 产生出动态分裂代码（如上一节所述），再修改其中部份程序。

不论动态分裂或静态分裂，分裂窗口都由CSplitterWnd 提供服务。动态分裂窗口的诞生是靠 CSplitterWnd::Create，静态分裂窗口的诞生则是靠 CSplitterWnd::CreateStatic。为了静态分裂，我们应该把上一节由 AppWizard 产生的函数代码改变如下：

```
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
CCreateContext* pContext)
{
    // 产生静态分裂窗口，横列为 1，纵行为 2。
    m_wndSplitter.CreateStatic(this, 1, 2);
    // 产生第一个窗口（标号 0,0）的 view 窗口。
    m_wndSplitter.CreateView(0, 0, RUNTIME_CLASS(CTextView),
    CSize(100, 0), pContext);
    // 产生第二个窗口（标号 0,1）的 view 窗口。
    m_wndSplitter.CreateView(0, 1, RUNTIME_CLASS(CBarView),
    CSize(0, 0), pContext);
}
```

这会产生如下的分裂窗口：

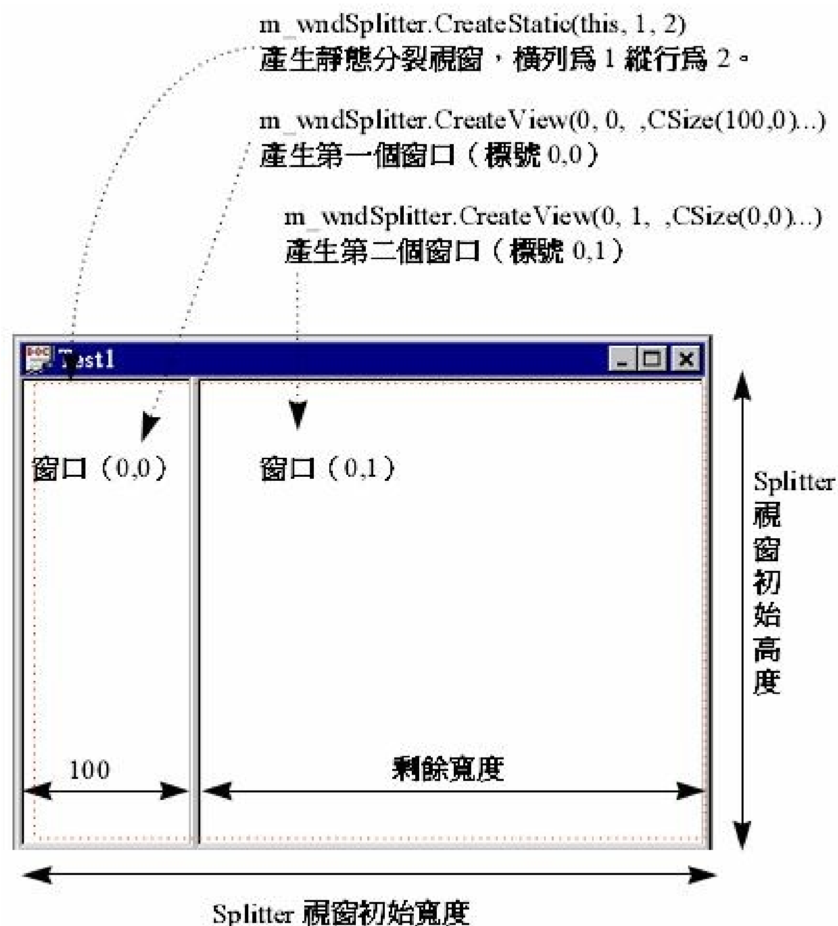
CreateStatic 和 CreateView

静态分裂用到两个 CSplitterWnd 成员函数：

CreateStatic：

这个函数的规格如下：

```
BOOL CreateStatic( CWnd* pParentWnd, int nRows, in nCols,
DWORD dwStyle = WS_CHILD | WS_VISIBLE,
UINT nID = AFX_IDW_PANE_FIRST );
```



第一个参数代表此分裂窗口之父窗口。第二和第三参数代表横列和纵行的个数。第四个参数是窗口风格，预设为 WS_CHILD | WS_VISIBLE，第五个同时也是最后一个参数代表窗口（也是一个窗口）的ID起始值。

CreateView

这个函数的规格如下：

```
virtual BOOL CreateView( int row, int col, CRuntimeClass* pViewClass,
SIZE sizeInit, CCreateContext* pContext );
```

第一和第二参数代表窗口的标号（从 0 起算）。第三参数是 View 类的 CRuntimeClass指针，你可以利用RUNTIME_CLASS宏（第 3 章和第 8 章提过）取此指针，也可以利用 OnCreateClient 的第二个参数CCreateContext* pContext 所储存的一个成员变量 m_pNewViewClass。你大概已经忘了这个变量吧，但我早提过它了，请看第 8 章的「CDocTemplate 管理 CDocument / CView / CFrameWnd」一节。所以，对于已在 CMultiDocTemplate 中登记过的 View 类，此处可以这么写：

```
// 产生第一个窗口（标号 0,0）的 view 窗口。
m_wndSplitter.CreateView(0, 0, RUNTIME_CLASS(CMyView),
CSize(100, 0), pContext);
```

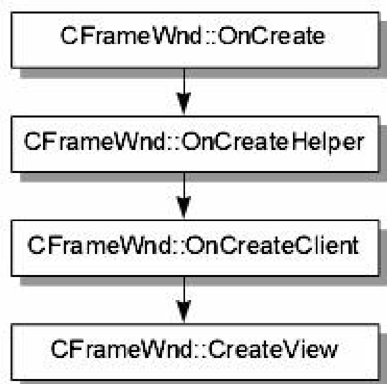
也可以这么写：

```
m_wndSplitter.CreateView(0, 0, pContext->m_pNewViewClass,
CSize(100, 0), pContext);
```

让我再多提醒你一些，第 8 章的「CDocTemplate 管理CDocument / CView / CFrameWnd」一节主要是说明当使用者打开一份文件，MFC 内部有关于 Document / View / Frame「三位一体」的动态生成过程。其中View 的动态生成是在 CFrameWnd::OnCreate被唤起后，经历一连串动作，最后才在 CFrameWnd::CreateView 中完成的：

而我们现在，为了分裂窗口，正在改写其中第三个虚函数 CFrameWnd::OnCreateClient呢！

好了，回过头来，CreateView 的第四参数是窗口的初始大小，CSize(100, 0) 表示窗口宽度为 100 个图素。高度倒是不为 0，对于横列为 1 的分裂窗口而言，窗口高度永远为窗口高度，Framework 并不理会你在CSize 中写了什么高度。至于第二个窗口的大小CSize(0, 0) 道理雷同，Framework 并不加理会其值，因为对于纵行为 2 的分裂窗口而言，右边窗口的宽度永远是窗口总宽度减去左边窗口的宽度。



程序进行中如果需要窗口的大小，只要在 OnDraw 函数（通常是这里需要）中这么写即可：

```
RECT rc; this->GetClientRect(&rc);
```

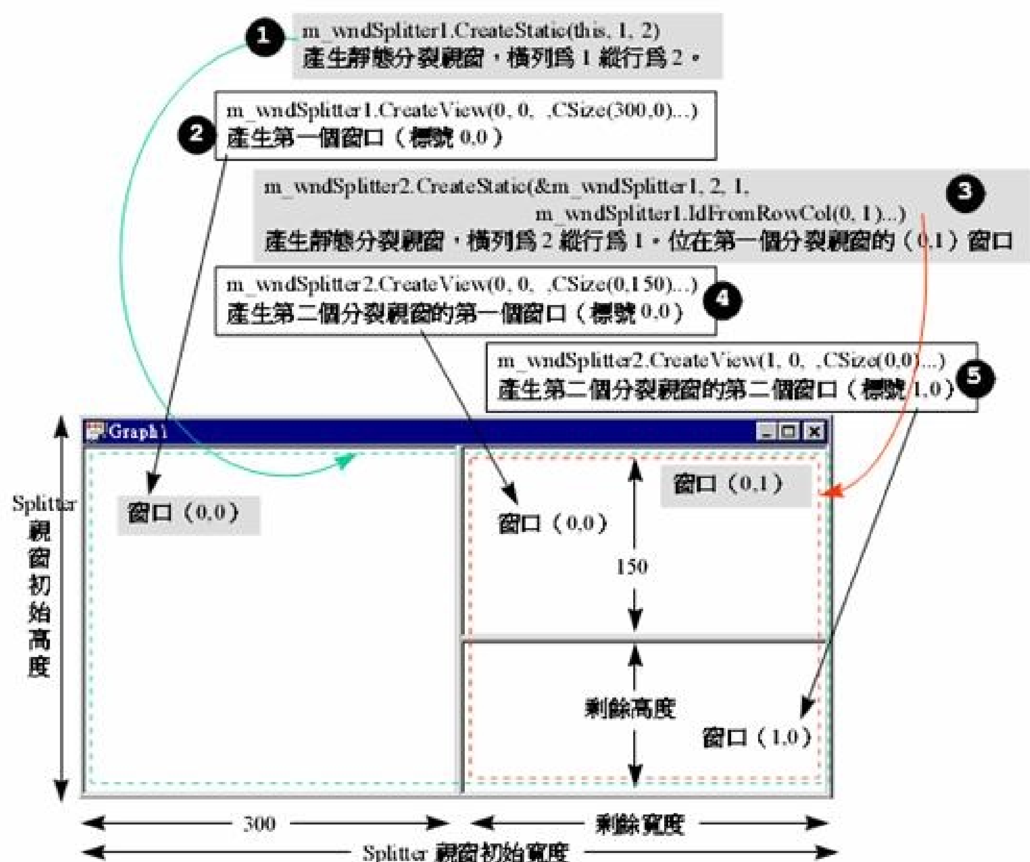
CreateView的第五参数是CCreateContext 指针。我们只要把 OnCreateClient 获得的第二个参数依样画葫芦地传下去就是了。

窗口的静态三叉分裂

分裂的方向可以无限延伸。我们可以把一个静态分裂窗口的窗口再做静态分裂，下面的程序代码展现了这种可能性：

```
// in header file
class CChildFrame : public CMDIChildWnd
{
    ...
protected:
    CSplitterWnd m_wndSplitter1;
    CSplitterWnd m_wndSplitter2;
public:
    // Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CChildFrame)
public:
    virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL
    ...
};
// in implementation file
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
CCreateContext* pContext)
{
    // 产生静态分裂窗口，横列为 1，纵行为 2。
    m_wndSplitter1.CreateStatic(this, 1, 2);
    // 产生分裂窗口的第一个窗口（标号 0,0）的 view 窗口。
    m_wndSplitter1.CreateView(0, 0, RUNTIME_CLASS(CTextView),
    CSize(300, 0), pContext);
    // 产生第二个分裂窗口，横列为 2，纵行为 1。位在第一个分裂窗口的（0,1）窗口
    m_wndSplitter2.CreateStatic(&m_wndSplitter1, 2, 1,
    WS_CHILD | WS_VISIBLE, m_wndSplitter1.IdFromRowCol(0, 1));
    // 产生第二个分裂窗口的第一个窗口（标号 0,0）的 view 窗口。
    m_wndSplitter2.CreateView(0, 0, RUNTIME_CLASS(CBarView),
    CSize(0, 150), pContext);
    // 产生第二个分裂窗口的第二个窗口（标号 1,0）的 view 窗口。
    m_wndSplitter2.CreateView(1, 0, RUNTIME_CLASS(CCurveView),
    CSize(0, 0), pContext);
    return TRUE;
}
```

这会产生如下的分裂窗口：



第二个分裂窗口的ID 起始值可由第一个分裂窗口的窗口之一获知（利用 IdFromRowCol 成员函数），一如上述程序代码中的动作。

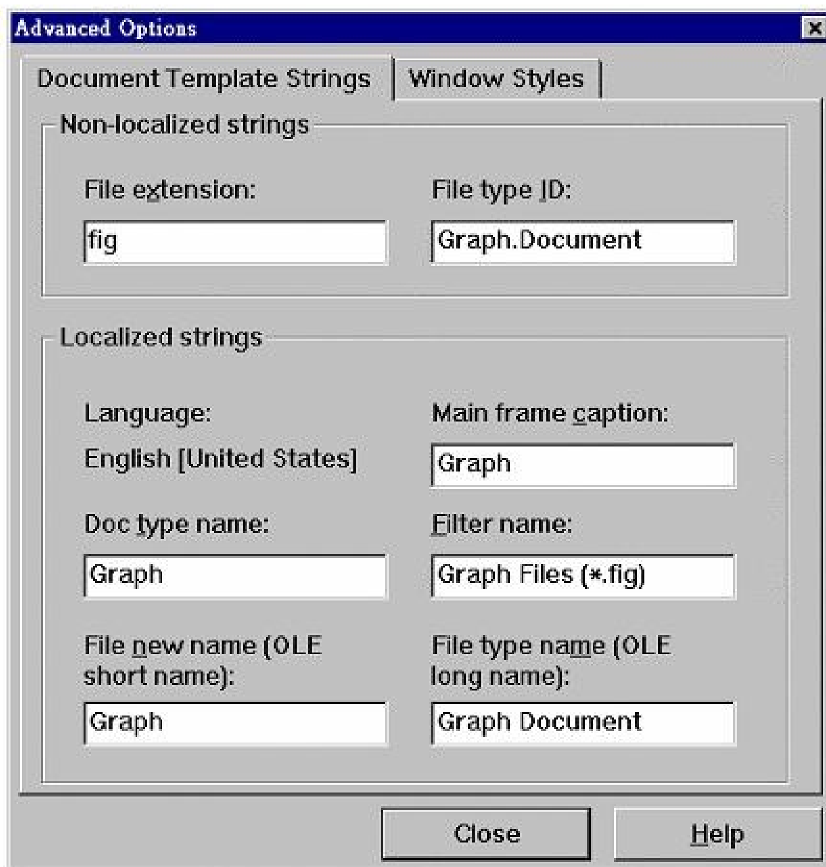
剩下的问题，就是如何设计许多个 View 类了。

Graph 范例程序

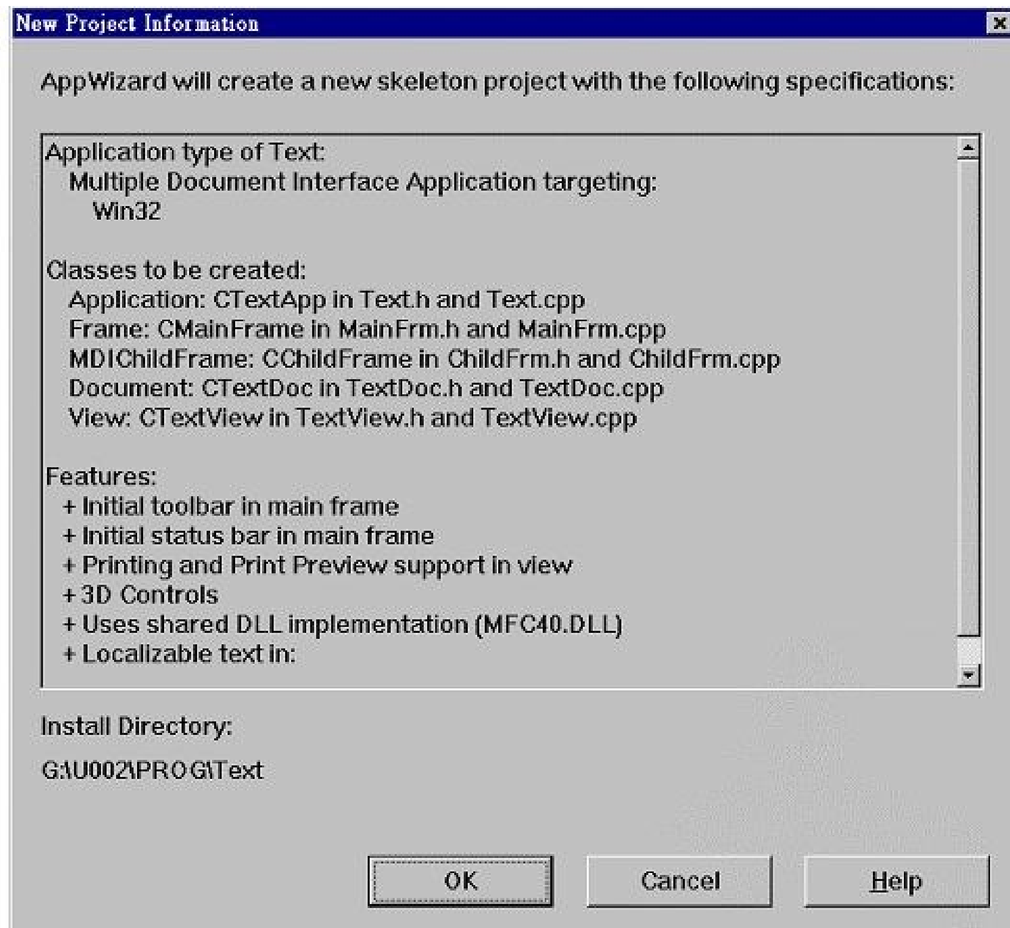
Graph 是一个具备静态三叉分裂能力的程序。左侧窗口以文字方式显示 10 笔整数数据，右上侧窗口显示该 10 笔数据的长条图，右下侧窗口显示对应的曲线图。

进行至这一章，相信各位对于工具的基本操作技术都已经都熟练了，这里我只列出Graph 程序的制作大纲：

- 进入AppWizard，制造一个Graph 项目。采用预设的选项，但在第四步骤的【Advanced】对话框的【Windows Styles】附页中，将【Use split window】致能（enabled）起来。并填写【Documents Template Strings】附页如下：



最后，AppWizard 给我们这样一份清单：



我们获得的主要类整理如下：

类	基类	文件	
CGraphApp	CWinApp	GRAPH.CPP	GRAPH.H
CMainFrame	CMDIFrameWnd	MAINFRM.CPP	MAINFRM.H
CChildFrame	CMDIChildWnd	CHILDFRM.CPP	CHILDFRM.H
CGraphDoc	CDocument	GRAPHDOC.CPP	GRAPHDOC.H
CGraphView	CView	GRAPHVIEW.CPP	GRAPHVIEW.H

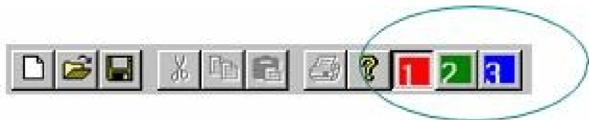
- 进入整合环境的Resource View窗口中，选择IDR_GRAPHTYPE选单，在【Window】之前加入一个【Graph Data】选单，并添加三个项目，分别是：

选单项目名称	识别代码（ID）	提示字符串
Graph Data&1	ID_GRAPH_DATA1	"Graph Data 1"
Graph Data&2	ID_GRAPH_DATA2	"Graph Data 2"
Graph Data&3	ID_GRAPH_DATA3	"Graph Data 3"

于是GRAPH.RC 的选单资源改变如下：

```
IDR_GRAPHTYPE MENU PRELOAD DISCARDABLE
BEGIN
...
POPUP "&Graph Data"
BEGIN
MENUITEM "Data&1", ID_GRAPH_DATA1
MENUITEM "Data&2", ID_GRAPH_DATA2
MENUITEM "Data&3", ID_GRAPH_DATA3
END
...
END
```

- 回到整合环境的Resource View 窗口，选择IDR_MAINFRAME 工具列，增加三个按钮，放在Help 按钮之后，并使用工具箱上的Draws Text 功能，为三个按钮分别涂上1, 2, 3 画面：



这三个按钮的IDs 采用先前新增的三个选单项目的IDs。

于是，GRAPH.RC 的工具列资源改变如下：

```
IDR_MAINFRAME TOOLBAR DISCARDABLE  16, 15
BEGIN
...
BUTTON ID_FILE_PRINT
BUTTON ID_APP_ABOUT
BUTTON ID_GRAPH_DATA1
BUTTON ID_GRAPH_DATA2
BUTTON ID_GRAPH_DATA3
END
```

- 进入ClassWizard，为新增的这些UI对象制作Message Map。由于这些命令项会影响到我们的Document 内容（当使用者按下Data1，我们必须为他准备一份相关数据；按下Data2，我们必须再为他准备一份相关数据），所以在CGraphDoc 中处理这些命令消息甚为合适：

UI 对象	Messages	消息处理例程
ID_GRAPH_DATA1	COMMAND	OnGraphData1
	UPDATE_COMMAND_UI	OnUpdateGraphData1
ID_GRAPH_DATA2	COMMAND	OnGraphData2
	UPDATE_COMMAND_UI	OnUpdateGraphData2
ID_GRAPH_DATA3	COMMAND	OnGraphData3
	UPDATE_COMMAND_UI	OnUpdateGraphData3

原始代码改变如下：

```
// in GRAPHDOC.H
class CGraphDoc : public CDocument
{
    ...
    // Generated message map functions
protected:
    //{{AFX_MSG(CGraphDoc)
    afx_msg void OnGraphData1();
    afx_msg void OnGraphData2();
    afx_msg void OnGraphData3();
    afx_msg void OnUpdateGraphData1(CCmdUI* pCmdUI);
    afx_msg void OnUpdateGraphData2(CCmdUI* pCmdUI);
    afx_msg void OnUpdateGraphData3(CCmdUI* pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
// in GRAPHDOC.CPP
BEGIN_MESSAGE_MAP(CGraphDoc, CDocument)
//{{AFX_MSG_MAP(CGraphDoc)
ON_COMMAND(ID_GRAPH_DATA1, OnGraphData1)
ON_COMMAND(ID_GRAPH_DATA2, OnGraphData2)
ON_COMMAND(ID_GRAPH_DATA3, OnGraphData3)
ON_UPDATE_COMMAND_UI(ID_GRAPH_DATA1, OnUpdateGraphData1)
ON_UPDATE_COMMAND_UI(ID_GRAPH_DATA2, OnUpdateGraphData2)
ON_UPDATE_COMMAND_UI(ID_GRAPH_DATA3, OnUpdateGraphData3)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

- 利用ClassWizard产生两个新类，做为三叉分裂窗口中的另两个窗口的View 类：

```
类名称      基类                文件

CTextView   CView               TEXTVIEW.CPP TEXTVIEW.H

CBarView    CView               BARVIEW.CPP BARVIEW.H
```

- 改写CChildFrame::OnCreateClient 函数如下（这是本节的技术重点）：

```
#include "stdafx.h"
#include "Graph.h"
#include "ChildFrm.h"
#include "TextView.h"
#include "BarView.h"
...
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
CCreateContext* pContext)
{
    // 产生静态分裂窗口，横列为 1，纵行为 2。
    m_wndSplitter1.CreateStatic(this, 1, 2);
    // 产生分裂窗口的第一个窗口（标号 0,0）的 view 窗口，采用 CTextView。
    m_wndSplitter1.CreateView(0, 0, RUNTIME_CLASS(CTextView),
    CSize(300, 0), pContext);
    // 产生第二个分裂窗口，横列为 2 纵行为 1。位在第一个分裂窗口的（0,1）窗口
    m_wndSplitter2.CreateStatic(&m_wndSplitter1, 2, 1,
    WS_CHILD|WS_VISIBLE, m_wndSplitter1.IdFromRowCol(0, 1));
    // 产生第二个分裂窗口的第一个窗口（标号 0,0）的 view 窗口，采用 CBarView。
    m_wndSplitter2.CreateView(0, 0, RUNTIME_CLASS(CBarView),
    CSize(0, 150), pContext);
    // 产生第二个分裂窗口的第二个窗口（标号 1,0）的 view 窗口，采用 CGraphView。
    m_wndSplitter2.CreateView(1, 0, pContext->m_pNewViewClass,
    CSize(0, 0), pContext);
    // 设定 active pane
    SetActiveView((CView*)m_wndSplitter1.GetPane(0,0));
    return TRUE;
}
```

为什么最后一次CreateView时我以pContext->m_pNewViewClass取代RUNTIME_CLASS(CGraphView)呢？后者当然也可以，但却因此必须含入CGraphView的声明；而如果你因为这个原因而含入GraphView.h 檔，又会产生三个编译错误，挺麻烦！

至此，Document 中虽然没有任何数据，但程序的 UI 已经完备，编译链接后可得以下执行画面：



修改CGraphDoc，增加一个整数数组m_intArray，这是真正存放数据的地方，我采用MFC内建的 `CArray<int,int>` ，为此，必须在STDAFX.H 中加上一行：

```
#include <afxtempl.h> // MFC templates
```

为了设定数组内容，我又增加了一个SetValue 成员函数，并且在【Graph Data】选单命令被执行时，为 m_intArray 设定不同的初值：

```
// in GRAPHDOC.H
class CGraphDoc : public CDocument
{
    ...
public:
    CArray<int,int> m_intArray;
public:
    void SetValue(int i0, int i1, int i2, int i3, int i4,
    int i5, int i6, int i7, int i8, int i9);
    ...
};
// in GRAPHDOC.CPP
CGraphDoc::CGraphDoc()
{
    SetValue(5, 10, 15, 20, 25, 78, 64, 38, 29, 9);
}
void CGraphDoc::SetValue(int i0, int i1, int i2, int i3, int i4,
int i5, int i6, int i7, int i8, int i9);
{
    m_intArray.SetSize(DATANUM, 0);
    m_intArray[0] = i0;
    m_intArray[1] = i1;
    m_intArray[2] = i2;
    m_intArray[3] = i3;
    m_intArray[4] = i4;
    m_intArray[5] = i5;
    m_intArray[6] = i6;
    m_intArray[7] = i7;
    m_intArray[8] = i8;
    m_intArray[9] = i9;
}
void CGraphDoc::OnGraphData1()
{
    SetValue(5, 10, 15, 20, 25, 78, 64, 38, 29, 9);
    UpdateAllViews(NULL);
}
void CGraphDoc::OnGraphData2()
{
    SetValue(50, 60, 70, 80, 90, 23, 68, 39, 73, 58);
    UpdateAllViews(NULL);
}
void CGraphDoc::OnGraphData3()
{
    SetValue(12, 20, 8, 17, 28, 37, 93, 45, 78, 29);
    UpdateAllViews(NULL);
}
void CGraphDoc::OnUpdateGraphData1(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_intArray[0] == 5);
}
void CGraphDoc::OnUpdateGraphData2(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_intArray[0] == 50);
}
void CGraphDoc::OnUpdateGraphData3(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_intArray[0] == 12);
}
```

各位看到，为了方便，我把m_intArray 的数据封装属性设为public 而非 private，检查「m_intArray 内容究竟是哪一份数据」所用的方法也非常粗糙，呀，不要非难我，重点不在这里呀！

- 在RESOURCE.H 文件中加上两个常数定义：

```
#define DATANUM   10
#define DATAMAX   100
```

- 修改CGraphView，在OnDraw 成员函数中取得Document，再透过Document对象指针取得整数数组，然后将10 笔数据的曲线图绘出：

```
#0001  void CGraphView::OnDraw(CDC* pDC)
#0002  {
#0003      CGraphDoc* pDoc = GetDocument();
#0004      ASSERT_VALID(pDoc);
#0005
#0006      int        cxDot,cxDotSpacing,cyDot, cxGraph,cyGraph, x,y, i;
#0007      RECT       rc;
#0008
#0009      CPen    pen (PS_SOLID, 1, RGB(255, 0, 0)); // red pen
#0010      CBrush brush(RGB(255, 0, 0));            // red brush
#0011      CBrush* pOldBrush = pDC->SelectObject(&brush);
#0012      CPen*   pOldPen = pDC->SelectObject(&pen);
#0013
#0014      cxGraph = 100;
#0015      cyGraph = DATAMAX;  // defined in resource.h
#0016
#0017      this->GetClientRect(&rc);
#0018      pDC->SetMapMode(MM_ANISOTROPIC);
#0019      pDC->SetWindowOrg(0, 0);
#0020      pDC->SetViewportOrg(10, rc.bottom-10);
#0021      pDC->SetWindowExt(cxGraph, cyGraph);
#0022      pDC->SetViewportExt(rc.right-20, -(rc.bottom-20));
#0023
#0024      // 我們希望圖形佔滿視窗的整個可用空間(以水平方向爲準)
#0025      // 並希望曲線點的寬度是點間距寬度的 1.2，
#0026      // 所以 (dot_spacing + dot_width) * num_datapoints = graph_width
#0027      // 亦即 dot_spacing * 3/2  * num_datapoints = graph_width
#0028      // 亦即 dot_spacing = graph_width / num_datapoints * 2/3
#0029
#0030      cxDotSpacing = (2 * cxGraph) / (3 * DATANUM);
#0031      cxDot = cxDotSpacing/2;
#0032      if (cxDot<3)  cxDot = 3;
#0033      cyDot = cxDot;
#0034
#0035      // 座標軸
#0036      pDC->MoveTo(0, 0);
#0037      pDC->LineTo(0, cyGraph);
#0038      pDC->MoveTo(0, 0);
#0039      pDC->LineTo(cxGraph, 0);
#0040
#0041      // 畫出資料點
#0042      pDC->SelectObject(::GetStockObject (NULL_PEN));
#0043      for (x=0+cxDotSpacing,y=0,i=0; i<DATANUM; i++,x+=cxDot+cxDotSpacing)
#0044          pDC->Rectangle(x, y+pDoc->m_intArray[i],
#0045                         x+cxDot, y+pDoc->m_intArray[i]-cyDot);
#0046
#0047     pDC->SelectObject(pOldBrush);
#0048     pDC->SelectObject(pOldPen);
#0049  }
```

- 修改CTextView 程序代码，在OnDraw成员函数中取得Document，再透过Document 对象指针取得整数数组，然后将10 笔数据以文字方式显示出来：

```
#0001  #include "stdafx.h"
#0002  #include "Graph.h"
#0003  #include "GraphDoc.h"
#0004  #include "TextView.h"
#0005  ...
#0006  void CTextView::OnDraw(CDC* pDC)
#0007  {
#0008       CGraphDoc* pDoc = (CGraphDoc*)GetDocument();
#0009
#0010       TEXTMETRIC tm;
#0011       int     x,y, cy, i;
#0012       char    sz[20];
#0013       pDC->GetTextMetrics(&tm);
#0014       cy = tm.tmHeight;
#0015       pDC->SetTextColor(RGB(255, 0, 0)); // red text
#0016       for (x=5,y=5,i=0; i<DATANUM; i++,y+=cy)
#0017      {
#0018           wsprintf (sz, "%d", pDoc->m_intArray[i]);
#0019           pDC->TextOut (x,y, sz, lstrlen(sz));
#0020      }
#0021  }
```

- 修改CBarView程序代码，在OnDraw 成员函数中取得Document，再透过Document 对象指针取得整数数组，然后将10 笔数据以长条图绘出：

```
#0001  #include "stdafx.h"
#0002  #include "Graph.h"
#0003  #include "GraphDoc.h"
#0004  #include "TextView.h"
#0005  ...
#0006  void CBarView::OnDraw(CDC* pDC)
#0007  {
#0008      CGraphDoc* pDoc = (CGraphDoc*)GetDocument();
#0009
#0010      int         cxBar,cxBarSpacing, cxGraph,cyGraph, x,y, i;
#0011      RECT        rc;
#0012
#0013      CBrush brush(RGB(255, 0, 0));  // red brush
#0014      CBrush* pOldBrush = pDC->SelectObject(&brush);
#0015      CPen pen(PS_SOLID, 1, RGB(255, 0, 0)); // red pen
#0016      CPen* pOldPen = pDC->SelectObject(&pen);
#0017
#0018      cxGraph = 100;
#0019      cyGraph = DATAMAX;  // defined in resource.h
#0020
#0021      this->GetClientRect(&rc);
#0022      pDC->SetMapMode(MM_ANISOTROPIC);
```

```
#0023      pDC->SetWindowOrg(0, 0);
#0024      pDC->SetViewportOrg(10, rc.bottom-10);
#0025      pDC->SetWindowExt(cxGraph, cyGraph);
#0026      pDC->SetViewportExt(rc.right-20, -(rc.bottom-20));
#0027
#0028      // 長條圖的條狀物之間距離是條狀物寬度的 1/3。
#0029      // 我們希望條狀物能夠填充視窗的整個可用空間。
#0030      // 所以 (bar_spacing + bar_width) * num_bars = graph_width
#0031      // 亦即  bar_width * 4/3  * num_bars = graph_width
#0032      // 亦即  bar_width = graph_width / num_bars * 3/4
#0033
#0034      cxBar = (3 * cxGraph) / (4 * DATANUM);
#0035      cxBarSpacing = cxBar/3;
#0036      if (cxBar<3)  cxBar=3;
#0037
#0038      // 座標軸
#0039      pDC->MoveTo(0, 0);
#0040      pDC->LineTo(0, cyGraph);
#0041      pDC->MoveTo(0, 0);
#0042      pDC->LineTo(cxGraph, 0);
#0043
#0044      // 長條圖
#0045      for (x=0+cxBarSpacing,y=0,i=0; i< DATANUM; i++,x+=cxBar+cxBarSpacing)
#0046          pDC->Rectangle(x, y, x+cxBar, y+pDoc->m_intArray[i]);
#0047
#0048    pDC->SelectObject(pOldPen);
#0049    pDC->SelectObject(pOldBrush);
#0050  }
```

- 如果你要令三个view 都有打印预视能力，必须在每一个view 类中改写以下三个虚函数：

```
virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
```

至于其函数内容，从 CGraphView 的同名函数中依样画葫芦拷贝一份过来即可。

- 本例不示范文件读写动作，所以CGraphDoc 没有改写Serialize 虚函数。

图13-3是Graph 程序的执行画面。

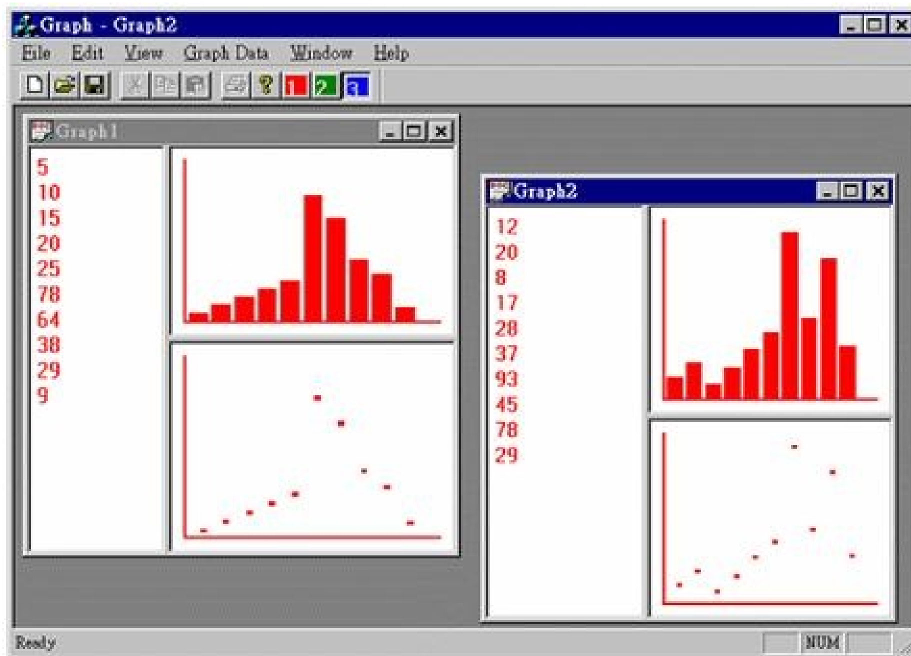


图13-3 Graph 执行画面。每当选按【File/New】（或工具列上的对应按钮）打开一份新文件，其内就有10笔整数数据。你可选按【Graph Data】（或工具列上的对应按钮）改变数据内容。

静态分裂窗口之观念整理

我想你已经从前述的OnCreateClient函数中认识了静态分裂窗口的相关函数。我可以用图 13-4 解释各个类的关系与运用。

基本上图 13-4 三个窗口可以视为三个完全独立的 view 窗口，有各自的类，以各自的方式显示数据。不过，数据倒是来自同一份 Document。试试看预视效果，你会发现，哪一个窗口为「作用中」，哪一个窗口的绘图动作就主宰预窗口口。你可以利用SetActivePane设定作用中的窗口，也可以调用 GetActivePane 获得作用中的窗口。但是，你会发现，从外观上很难看出哪一个窗口是「作用中的」窗口。

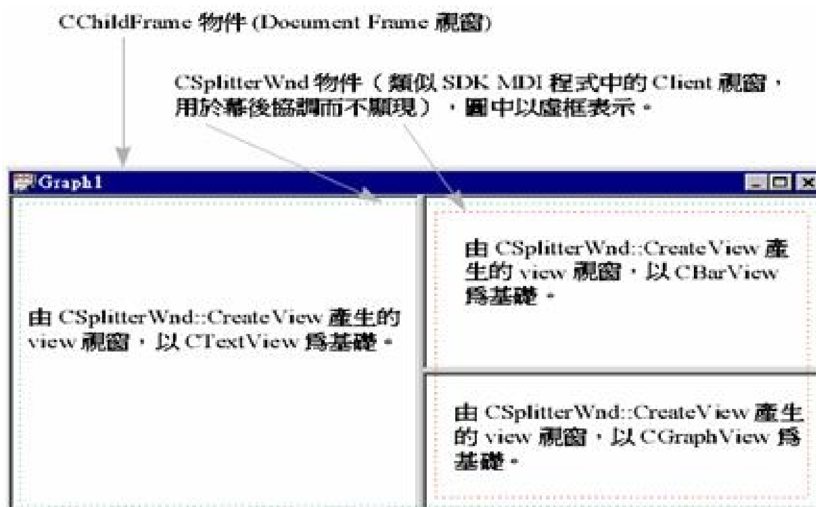


图13-4 静态分裂窗口的类运用 (以Graph为例)

同源子窗口

虽然我说静态分裂窗口的窗口可视为完全独立的 view 窗口，但毕竟它们不是！它们还框在一个大窗口中。如果你不喜欢分裂窗口（谁知道呢，当初我也不太喜欢），我们来试点新鲜的。

点子是从【Window/New Window】开始。这个选单项目令 Framework 为我们做出目前作用中的view窗口的另一份拷贝。如果我们能够知道 Framework 是如何动作，是不是可以引导它使用另一个view类，以不同的方式表现同一份数据？

这就又有偷窥原始代码的需要了。MFC 并没有提供正常的管道让我们这么做，我们需要MFC原始代码。

CMDIFrameWnd::OnWindowNew

如果你想在程序中设计中断点，一步一步找出【Window/New Window】的动作，就像我在第12章对付OnFilePrint和OnFilePrintPreview 一样，那么你会发现没有着力点，因为AppWizard并不会做出像这样的消息映射表格：

```
BEGIN_MESSAGE_MAP(CScribbleView, CScrollView)
...
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

你根本不知道【Window/New Window】这个命令流到哪里去了。第 7 章的「标准选单 File/Edit / View / Window / Help」一节，也曾说过这个命令项是属于「与 Framework 预有关联型」的。

那么我如何察其流程？1/3 用猜的，1/3 靠字符串搜寻工具 GREP（第 8 章介绍过），1/3 靠勤读书。然后我发现，【New Window】命令流到 CMDIFrameWnd::OnWindowNew 去了。

图 13-5 是其原始代码（MFC 4.0 的版本）。

```
#0001 void CMDIFrameWnd::OnWindowNew()
#0002 {
#0003     CMDIChildWnd* pActiveChild = MDIGetActive();
#0004     CDocument* pDocument;
#0005     if (pActiveChild == NULL ||
#0006       (pDocument = pActiveChild->GetActiveDocument()) == NULL)
#0007     {
#0008         TRACE0("Warning: No active document for WindowNew command.\n");
#0009         AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0010         return;     // command failed
#0011     }
#0012
#0013     // otherwise we have a new frame !
#0014     CDocTemplate* pTemplate = pDocument->GetDocTemplate();
#0015     ASSERT_VALID(pTemplate);
#0016     CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument, pActiveChild);
#0017     if (pFrame == NULL)
#0018     {
#0019         TRACE0("Warning: failed to create new frame.\n");
#0020         return;     // command failed
#0021     }
#0022
#0023     pTemplate->InitialUpdateFrame(pFrame, pDocument);
#0024 }
```

图13-5 CMDIFrameWnd::OnWindowNew原始代码（in WINMDI.CPP）

我们的焦点放在CMDIFrameWnd::OnWindowNew 函数的第14 行，该处取得我们在InitInstance 函数中做好的 Document Template，而你知道，Document Template 中记录有View 类。好，如果我们能够另准备一个崭新的View类，有着不同的OnDraw 显示方式，并再准备好另一份Document Template，记录该新的View 类，然后改变图13-5 的第14行，让它使用这新的Document Template，大功成矣。

当然，我们绝不是要去改MFC原始代码，而是要改写虚函数OnWindowNew，使为我们所用。这很简单，我只要把【Window / New Window】命令项改变名称，例如改为【Window / New Hex Window】，然后为它撰写命令处理函数，函数内容完全仿照图 13-5，但把第14 行改设定为新的Document Template 即可。

Text 范例程序

Text 程序提供【Window / New Text Window】和【Window / New Hex Window】两个新的选单命令项目，都可以产生出 view 窗口，一个以ASCII型式显示数据，一个以Hex型式显示数据，数据来自同一份Document。

以下Text程序的是制作过程：

- 进入AppWizard，制造一个Text 项目，采用各种预设的选项。获得的主要类如下：

类	基类	文件

CTextApp	*CWinApp*	TEXT.CPP	TEXT.H
CMainFrame	*CMDIFrameWnd*	MAINFRM.CPP	MAINFRM.H
CChildFrame	*CMDIChildWnd*	CHILDFRM.CPP	CHILDFRM.H
CTextDoc	*CDocument*	TEXTDOC.CPP	TEXTDOC.H
CTextView	*CView*	TEXTVIEW.CPP	TEXTVIEW.H

- 进入整合环境中的Resource View 窗口，选择IDR_TEXTTYPE选单，在【Window】选单中加入两个新命令项：

```
命令项目名称     识别代码（ID）     提示字符串
New Text Window ID_WINDOW_TEXT  New a Text Window with Active Document
New Hex Window  ID_WINDOW_HEX   New a Hex Window with Active Document
```

- 再在Resource View 窗口中选择IDR_MAINFRAME 工具列，增加两个按钮，安排在Help按钮之后：



这两个按钮分别对应于新添加的两个选单命令项目。

- 进入ClassWizard，为两个UI对象制作Message Map。这两个命令消息并不会影响Document内容（不像上一节的GRAPH 例那样），我们在CMainFrame中处理这两个命令消息颇为恰当。

```
UI 对象           消息          消息处理例程
ID_WINDOW_TEXT  COMMAND         OnWindowText
ID_WINDOW_HEX   COMMAND         OnWindowHex
```

- 利用ClassWizard 产生一个新类，准备做为同源子窗口的第二个View类：

```
类名称             基类          文件
CHexView              CView             HEXVIEW.CPP HEXVIEW.H
```

- 修改程序代码，分别为两个view类都做出对应的Docment Template：

```
// in TEXT.H
class CTextApp : public CWinApp
{
public:
       CMultiDocTemplate* m_pTemplateTxt;
       CMultiDocTemplate* m_pTemplateHex;
       ...
```

```
// in TEXT.H
class CTextApp : public CWinApp
{
public:
        CMultiDocTemplate* m_pTemplateTxt;
        CMultiDocTemplate* m_pTemplateHex;
        ...
        public:
        virtual BOOL InitInstance();
        virtual int  ExitInstance();
        ...
};

// in TEXT.CPP
...
#include "TextView.h"
#include "HexView.h"
...
BOOL CTextApp::InitInstance()
{
        ...
        CMultiDocTemplate* pDocTemplate;
        pDocTemplate = new CMultiDocTemplate(
                IDR_TEXTTYPE,
                RUNTIME_CLASS(CTextDoc),
                RUNTIME_CLASS(CChildFrame), // custom MDI child frame
                RUNTIME_CLASS(CTextView));
        AddDocTemplate(pDocTemplate);

m_pTemplateTxt = new CMultiDocTemplate(
        IDR_TEXTTYPE,
        RUNTIME_CLASS(CTextDoc),
        RUNTIME_CLASS(CChildFrame), // custom MDI child frame
        RUNTIME_CLASS(CTextView));

m_pTemplateHex = new CMultiDocTemplate(

                IDR_TEXTTYPE,
                RUNTIME_CLASS(CTextDoc),
                RUNTIME_CLASS(CChildFrame), // custom MDI child frame
                RUNTIME_CLASS(CHexView));
        ...
}

int CTextApp::ExitInstance()
{
        delete m_pTemplateTxt;
        delete m_pTemplateHex;
        return CWinApp::ExitInstance();
}
```

- 修改CTextDoc 程序代码，添加成员变量。Document 的数据是10 笔字符串：

```
// in TEXTDOC.H
class CTextDoc : public CDocument
{
public:
        CStringArray m_stringArray;
...
};
```

```
// in TEXTDOC.CPP
BOOL CTextDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    m_stringArray.SetSize(10);
    m_stringArray[0] = "If you love me let me know,  ";
    m_stringArray[1] = "if you don't then let me go, ";
    m_stringArray[2] = "I can take another minute    ";
    m_stringArray[3] = "  of day without you in it.  ";
    m_stringArray[4] = "                             ";
    m_stringArray[5] = "If you love me let it be,    ";
    m_stringArray[6] = "if you don't then set me free";
    m_stringArray[7] = "...                          ";
    m_stringArray[8] = "SORRY, I FORGET IT!          ";
    m_stringArray[9] = "          J.J.Hou 1995.03.22  19:26";

    return TRUE;
}
```

- 修改CTextView::OnDraw 函数代码，在其中取得Document对象指针，然后把文字显现出来：

```
// in TEXTVIEW.CPP
void CTextView::OnDraw(CDC* pDC)
{
        CTextDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        int i, j, nHeight;
        TEXTMETRIC tm;

        pDC->GetTextMetrics(&tm);
        nHeight = tm.tmHeight + tm.tmExternalLeading;

        j = pDoc->m_stringArray.GetSize();
        for (i = 0; i < j; i++) {
            pDC->TextOut(10, i*nHeight, pDoc->m_stringArray[i]);
        }
}
```

- 修改CHexView程序代码，在OnDraw函数中取得Document对象指针，把ASCII 转换为Hex代码，再以文字显示出来：

```
#0001  #include "stdafx.h"
#0002  #include "Text.h"
#0003  #include "TextDoc.h"
#0004  #include "HexView.h"
#0005  ...
#0006  void CHexView::OnDraw(CDC* pDC)
#0007  {
#0008          // CDocument* pDoc = GetDocument();
#0009          CTextDoc* pDoc = (CTextDoc*)GetDocument();
#0010
#0011          int        i, j, k, l, nHeight;
#0012          long       n;
#0013          char       temp[10];
#0014          CString    Line;
#0015          TEXTMETRIC tm;
#0016
#0017          pDC->GetTextMetrics(&tm);
#0018          nHeight = tm.tmHeight + tm.tmExternalLeading;
#0019
#0020          j = pDoc->m_stringArray.GetSize();

#0021          for(i = 0; i < j; i++) {
#0022              wsprintf(temp, "%02x    ", i);
#0023              Line = temp;
#0024              l = pDoc->m_stringArray[i].GetLength();
#0025              for(k = 0; k < l; k++) {
#0026                  n = pDoc->m_stringArray[i][k] & 0x00FF;
#0027                  wsprintf(temp, "%02lx ", n);
#0028                  Line += temp;
#0029              }
#0030              pDC->TextOut(10, i*nHeight, Line);
#0031          }
#0032  }
```

- 定义CMainFrame的两个命令处理例程：OnWindowText和OnWindowHex，使选单命令项目和工具列按钮得以发挥效用。函数内容直接拷贝自图 13-5，只要修改其中第14 行即可。这两个函数是本节的技术重点。

```
#0001  void CMainFrame::OnWindowText()
#0002  {
#0003      CMDIChildWnd* pActiveChild = MDIGetActive();
#0004      CDocument* pDocument;
#0005      if (pActiveChild == NULL ||
#0006         (pDocument = pActiveChild->GetActiveDocument()) == NULL)
#0007      {
#0008          TRACE0("Warning: No active document for WindowNew command\n");
#0009          AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0010          return;     // command failed
#0011      }
#0012
#0013      // otherwise we have a new frame!
#0014      CDocTemplate* pTemplate = ((CTextApp*) AfxGetApp())->m_pTemplateTxt;
#0015      ASSERT_VALID(pTemplate);
#0016      CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument, pActiveChild);
#0017      if (pFrame == NULL)
#0018      {
#0019          TRACE0("Warning: failed to create new frame\n");
#0020          AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0021          return;     // command failed
#0022      }
#0023
#0024      pTemplate->InitialUpdateFrame(pFrame, pDocument);
#0025  }
#0026
#0027  void CMainFrame::OnWindowHex()
#0028  {
#0029      CMDIChildWnd* pActiveChild = MDIGetActive();
#0030      CDocument* pDocument;
#0031      if (pActiveChild == NULL ||
#0032         (pDocument = pActiveChild->GetActiveDocument()) == NULL)
#0033      {
#0034          TRACE0("Warning: No active document for WindowNew command\n");
#0035          AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0036          return;     // command failed
#0037      }
#0038
#0039      // otherwise we have a new frame!
#0040      CDocTemplate* pTemplate = ((CTextApp*) AfxGetApp())->m_pTemplateHex;
#0041      ASSERT_VALID(pTemplate);
#0042      CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument, pActiveChild);
#0043      if (pFrame == NULL)
#0044      {
#0045          TRACE0("Warning: failed to create new frame\n");
#0046          AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0047          return;     // command failed
#0048      }
#0049
#0050      pTemplate->InitialUpdateFrame(pFrame, pDocument);
#0051  }
```

如果你要两个view 都有打印预视的能力，必须在CHexView 中改写下面三个虚函数，至于它们的内容，可以依样画葫芦地从CTextView 的同名函数中拷贝一份过来：

```
// in HEXVIEW.H
class CHexView : public CView
{
...
// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CHexView)
        protected:
        virtual void OnDraw(CDC* pDC);      // overridden to draw this view
        protected:
        virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
        virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
        virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
        //}}AFX_VIRTUAL
...
};
```

```
// in HEXVIEW.CPP
BEGIN_MESSAGE_MAP(CHexView, CView)
        //{{AFX_MSG_MAP(CHexView)
                // NOTE - the ClassWizard will add and remove mapping macros here.
        //}}AFX_MSG_MAP
        // Standard printing commands
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CTextView printing

BOOL CHexView::OnPreparePrinting(CPrintInfo* pInfo)
{
        // default preparation
        return DoPreparePrinting(pInfo);
}

void CHexView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
        // TODO: add extra initialization before printing
}

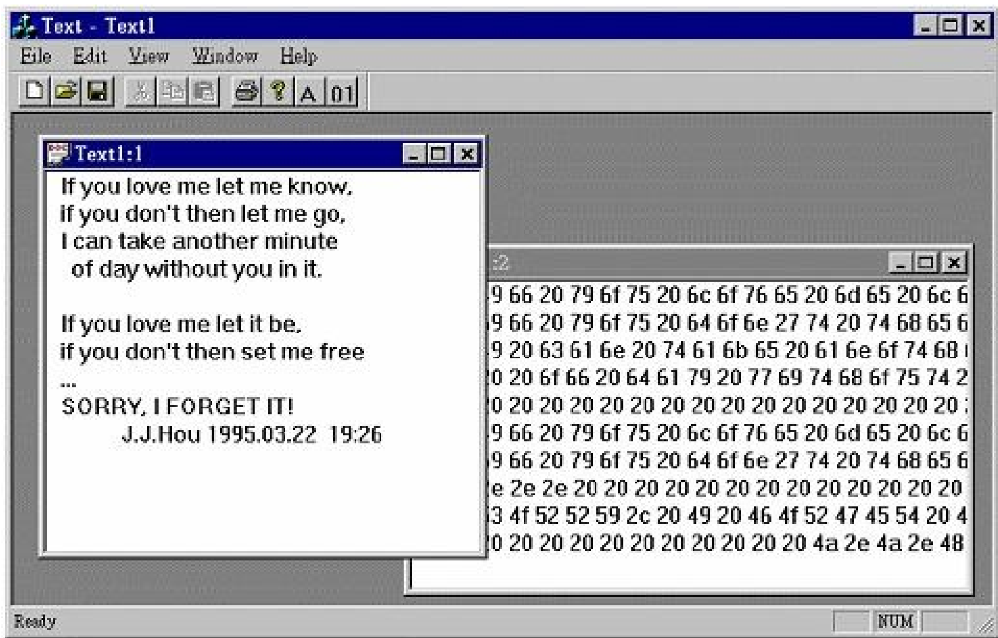
void CHexView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
        // TODO: add cleanup after printing
}
```

本例并未示范Serialization 动作。

非制式作法的缺点

既然是走后门，就难保哪一天出问题。如果MFC的版本变动，CMDIFrameWnd::OnWindowNew 内容改了，你就得注意本节这个方法还能适用否。

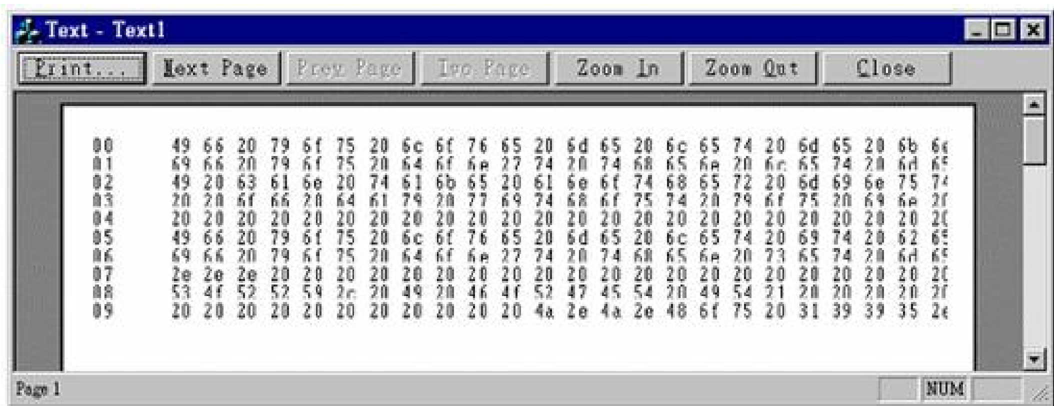
下面是Text 程序的执行画面。我先开启一个Text 窗口，再选按【Window/New Hex Window】或工具列上的对应按钮，开启另一个Hex 窗口。两个View 窗口以不同的方式显示同一份文件数据。



当你选按【File/Preview】命令项，哪一个窗口为 active 窗口，那个窗口的内容就出现在预视画面中。以下是Text窗口的打印预视画面：



以下是Hex窗口的打印预视画面：



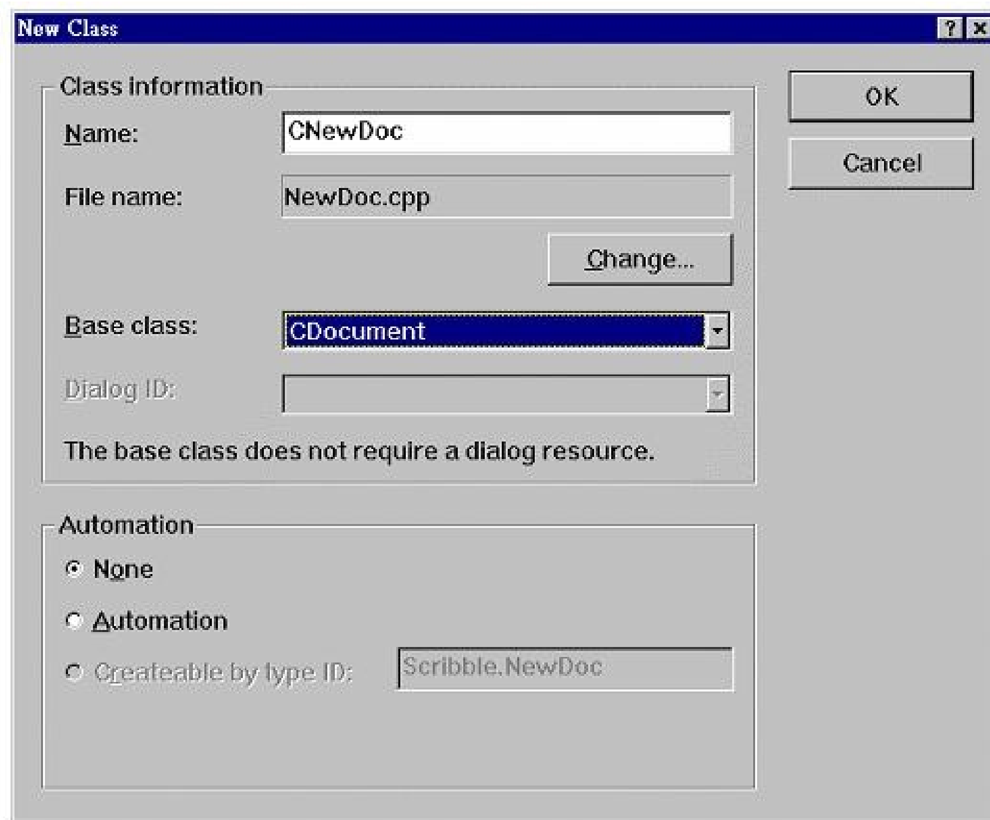
多重文件

截至目前，我所谈的都是如何以不同的方式在不同的窗口中显示同一份文件数据。如果我想写那种「多功能」软件，必须支持许多种文件类型，该怎么办？

就以前一节的 Graph 程序为基础，继续我们的探索。Graph 的文件类型原本是一个整数数组，数量有10笔。我想在上面再多支持一种功能：文字编辑能力。

新的Document类

首先，我应该利用ClassWizard 新添一个Document 类，并以CDocument 为基础。启动ClassWizard，选择【Member Variables】附页，按下【Add Class...】钮，出现对话框，填写如下：



下面是ClassWizard为我们做出来的代码：

```
#0001 // NewDoc.cpp : implementation file
#0002 //
#0003
#0004 #include "stdafx.h"
#0005 #include "Graph.h"
#0006 #include "NewDoc.h"
#0007
#0008 #ifdef _DEBUG
#0009 #define new DEBUG_NEW
#0010 #undef THIS_FILE
#0011 static char THIS_FILE[] = __FILE__;
#0012 #endif
#0013
#0014 /////////////////////////////////////////////////////////////////////////////
#0015 // CNewDoc
#0016
#0017 IMPLEMENT_DYNCREATE(CNewDoc, CDocument)
#0018
#0019 CNewDoc::CNewDoc()
#0020 {
#0021 }
#0022
#0023 BOOL CNewDoc::OnNewDocument()
#0024 {
#0025         if (!CDocument::OnNewDocument())
#0026                 return FALSE;
#0027         return TRUE;
#0028 }
#0029
#0030 CNewDoc::~CNewDoc()
#0031 {
#0032 }
#0033
#0034
#0035 BEGIN_MESSAGE_MAP(CNewDoc, CDocument)
#0036         //{{AFX_MSG_MAP(CNewDoc)
#0037         // NOTE - the ClassWizard will add and remove mapping macros here.
#0038         //}}AFX_MSG_MAP
#0039 END_MESSAGE_MAP()
#0040
#0041 /////////////////////////////////////////////////////////////////////////////
#0042 // CNewDoc diagnostics
#0043
#0044 #ifdef _DEBUG
#0045 void CNewDoc::AssertValid() const
#0046 {
#0047         CDocument::AssertValid();
#0048 }
#0049
#0050 void CNewDoc::Dump(CDumpContext& dc) const
#0051 {
```

```
#0052          CDocument::Dump(dc);
#0053  }
#0054  #endif //_DEBUG
#0055
#0056  /////////////////////////////////////////////////////////////////////////////
#0057  // CNewDoc serialization
#0058
#0059  void CNewDoc::Serialize(CArchive& ar)
#0060  {
#0061          if (ar.IsStoring())
#0062          {
#0063                  // TODO: add storing code here
#0064          }
#0065          else
#0066          {
#0067                  // TODO: add loading code here
#0068          }
#0069
#0070          // CEditView contains an edit control which handles all serialization
#0071          ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
#0072  }
#0073
#0074  /////////////////////////////////////////////////////////////////////////////
#0075  // CNewDoc commands
```

注：阴影中的这两行代码（#0070 和 #0071）不是 ClassWizard 产生的，是我自己加的，提前与你见面。稍后我会解释为什么加这两行。

新的 Document Template

然后，我应该为此新的文件类型产生一个 Document Template，并把它加到系统所维护的 DocTemplate 串列中。注意，为了享受现成的文字编辑能力，我选择CEditView 做为与此 Document 搭配之View 类。还有，由于 CChildFrame 已经因为第一个文件类型Graph 的三叉静态分裂而被我们改写了 OnCreateClient 函数，已不再适用于这第二个文件类型（NewDoc），所以我决定直接采用 CMDIChildWnd 做为 NewDoc 文件类型的 MDIChild Frame 窗口：

```
#include "stdafx.h"
#include "Graph.h"
#include "MainFrm.h"
#include "ChildFrm.h"
#include "GraphDoc.h"
#include "GraphView.h"
#include "NewDoc.h"
...
BOOL CGraphApp::InitInstance()
{
        ...
        CMultiDocTemplate* pDocTemplate;
        pDocTemplate = new CMultiDocTemplate(
                IDR_GRAPHTYPE,
                RUNTIME_CLASS(CGraphDoc),
                RUNTIME_CLASS(CChildFrame), // custom MDI child frame

                RUNTIME_CLASS(CGraphView));
        AddDocTemplate(pDocTemplate);

        pDocTemplate = new CMultiDocTemplate(
                IDR_NEWTYPE,
                RUNTIME_CLASS(CNewDoc),
                RUNTIME_CLASS(CMDIChildWnd), // use directly
                RUNTIME_CLASS(CEditView));
        AddDocTemplate(pDocTemplate);
        ...
}
```

CMultiDocTemplate 的第一个参数（resource ID）也不能再延用Graph 文件类型所使用的IDR_GRAPHTYPE了。要知道，这个ID值关系非常重大。我们得自行设计一套适用于NewDoc文件类型的UI系统出来（包括选单、工具列、文件存取对话框的内容、文件图标、窗口标题...）。

怎么做？第 7 章的深入讨论将在此开花结果！请务必回头复习复习「Document Template的意义」一节，我将直接动作，不再多做说明。

新的UI系统

下面就是为了这新的NewDoc文件类型所对应的UI系统，新添的文件内容（没有什么好工具可以帮忙，一般文字编辑器的copy/paste 最快）：

```
// in RESOURCE.H
#define IDD_ABOUTBOX                    100
#define IDR_MAINFRAME                   128
#define IDR_GRAPHTYPE                   129
#define IDR_NEWTYPE                     130
...

// in GRAPH.RC
IDR_NEWTYPE  ICON   DISCARDABLE   "res\\NewDoc.ico"   // 此 icon 需自行備妥

IDR_NEWTYPE MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
    MENUITEM "&New\tCtrl+N",                ID_FILE_NEW
    MENUITEM "&Open...\tCtrl+O",            ID_FILE_OPEN
    MENUITEM "&Close",                      ID_FILE_CLOSE
    MENUITEM "&Save\tCtrl+S",               ID_FILE_SAVE
    MENUITEM "Save &As...",                 ID_FILE_SAVE_AS
    MENUITEM SEPARATOR
    MENUITEM "&Print...\tCtrl+P",           ID_FILE_PRINT
    MENUITEM "Print Pre&view",              ID_FILE_PRINT_PREVIEW
    MENUITEM "P&rint Setup...",             ID_FILE_PRINT_SETUP
    MENUITEM SEPARATOR
    MENUITEM "Recent File",                 ID_FILE_MRU_FILE1, GRAYED
    MENUITEM SEPARATOR
    MENUITEM "E&xit",                       ID_APP_EXIT
END
POPUP "&Edit"
BEGIN
    MENUITEM "&Undo\tCtrl+Z",               ID_EDIT_UNDO
    MENUITEM SEPARATOR
    MENUITEM "Cu&t\tCtrl+X",                ID_EDIT_CUT
    MENUITEM "&Copy\tCtrl+C",               ID_EDIT_COPY
    MENUITEM "&Paste\tCtrl+V",              ID_EDIT_PASTE
END
POPUP "&View"
BEGIN
    MENUITEM "&Toolbar",                    ID_VIEW_TOOLBAR
    MENUITEM "&Status Bar",                 ID_VIEW_STATUS_BAR
END
POPUP "&Window"
BEGIN
    MENUITEM "&New Window",                 ID_WINDOW_NEW
    MENUITEM "&Cascade",                    ID_WINDOW_CASCADE
    MENUITEM "&Tile",                       ID_WINDOW_TILE_HORZ
    MENUITEM "&Arrange Icons",              ID_WINDOW_ARRANGE
    MENUITEM "S&plit",                      ID_WINDOW_SPLIT
END
```

```
   POPUP "&Help"
   BEGIN
      MENUITEM "&About Graph...",          ID_APP_ABOUT
   END
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME "Graph"
  IDR_GRAPHTYPE "Graph\nGraph\nGraph\nGraph Files
                 (*.fig)\n.FIG\nGraph.Document\nGraph Document"
```

```
  IDR_NEWTYPE   "NewDoc\nNewDoc\nNewDoc\nNewDoc Files
                 (*.txt)\n.TXT\nNewDoc.Document\nNewDoc Document"
END
```

新文件的文件读写动作

你大概还没有忘记，第 7 章最后曾经介绍过，当我们在 AppWizard 中选择 CEditView（而不是CView）作为我们的View类基础时，AppWizard 会为我们在 CMyDoc::Serialize函数中放入这样的代码：

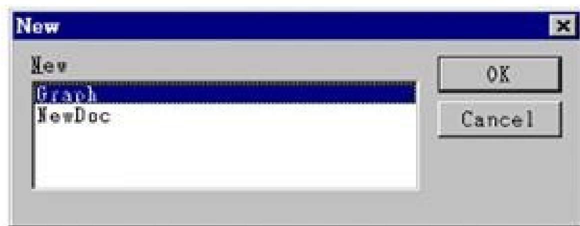
```
void CMyDoc::Serialize(CArchive& ar)
{
    // CEditView contains an edit control which handles all serialization
    ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
}
```

当你使用CEditView，编辑器窗口所承载的文字是放在 Edit 控制组件自己的一个内存区块中，而不是切割到Document 中。所以，文件的文件读写动作只要调用CEditView 的SerializeRaw函数即可。

为了这NewDoc文件类型能够读写文件，我们也依样画葫芦地把上一段代码阴影部份加到Graph程序新的Document类去：

```
void CNewDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
    // CEditView contains an edit control which handles all serialization
    ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
}
```

现在一切完备，重新编辑链接并执行。一开始，由于InitInstance 函数会自动为我们New一个新文件，而Graph 程序不知道该New 哪一种文件类型才好，所以会给我们这样的对话框：



往后每一次选按【File/New】，都会出现上述对话框。

以下是我们打开Graph 文件和NewDoc 文件各一份的画面。注意，当 active 窗口是NewDoc 文件，工具列上属于 Graph 文件所用的最后三个按钮是不起作用的：



以下是【Open】对话框（用来开文件）。注意，文件有 .fig 和 .txt 和 . 三种选择：



这个新的 Graph 版本放在书附光盘片的 \GRAPH2.13 目录中。

第 14 章 MFC 多线程程序设计

线程（thread），是线程（thread of execution）的简单称呼。"Thread" 这个字的原意是「线」。中文字里头的「线程」也有「线」的意思，所以我采用「线程」、「线程」这样的中文名称。如果你曾经看过「多线」这个名词，其实就是本章所谓的「多线程」。

我曾经在第 1 章以三两个小节介绍 Win32 环境下的进程与线程观念，并且以程序直接调用 CreateThread 的形式，示范了几个 Win32 小例子。现在我要更进一步从操作系统的层面谈谈线程的学理基础，然后带引各位看看 MFC 对于「线程」支持了什么样的类。然后，实际写个 MFC多线程程序。

从操作系统层面看线程

书籍推荐：如果要从操作系统层面来了解线程，Matt Pietrek 的 Windows 95 System Programming SECRETS（Windows 95 系统程序设计大奥秘/侯俊杰译/旗标出版）无疑是最佳知识来源。Matt 把操作系统核心模块（KERNEL32.DLL）中用来维护线程生存的数据结构都挖掘出来，非常详尽。这是对线程的最基础认识，直达其灵魂深处。

你已经知道，CreateThread 可以产生一个线程，而「线程」的本体就是 CreateThread 第 3 个参数所指定的一个函数（一般我们称之为「线程函数」）。这个函数将与目前的「执行事实」同时并行，成为另一个「执行事实」。线程函数的执行期，也就是该线程的生命期。

操作系统如何造成这种多任务并行的现象？线程对于操作系统的意义到底是什么？系统如何维护许多个线程？线程与其父亲大人（进程）的关系如何维持？CPU 只有一个，线程却有好几个，如何摆平优先权与排程问题？这些疑问都可以在下面各节中获得答案。

三个观念：模块、进程、线程

试着回答这个问题：进程（process）是什么？给你一分钟时间。

```
z z z z z...
```

你的回答可能是：『一个可执行文件执行起来，就是一个进程』。唔，也不能算错。但能不能够有更具体的答案？再问你一个问题：模块（module）是什么？可能你的回答还是：『一个可执行文件执行起来，就是一个模块』。这也不能够算错。但是你明明知道，模块不等于进程。KERNEL32 DLL 是一个模块，但不是一个进程；Scribble EXE 是一个模块，也是一个进程。

我们需要更具体的数据，更精准的答案。

如果我们能够知道操作系统如何看待模块和进程，就能够给出具体的答案了。一段可执行的程序（包括 EXE 和 DLL），其程序代码、数据、资源被加载到内存中，由系统建置一个数据结构来管理它，就是一个模块。这里所说的数据结构，名为 Module Database（MDB），其实就是PE格式中的PE表头，你可以从 WINNT.H 档中找到一个IMAGE_NT_HEADER 结构，就是它。

好，解释了模块，那么进程是什么？这就比较抽象一点了。这样说，进程就是一大堆拥有权（ownership）的集合。进程拥有地址空间（由 memory context 决定）、动态配置而来的内存、文件、线程、一系列的模块。操作系统使用一个所谓的 Process Database（PDB）数据结构，来记录（管理）它所拥有的一切。

线程呢？线程是什么？进程主要表达「拥有权」的观念，线程则主要表达模块中的程序代码的「执行事实」。系统也是以一个特定的数据结构（Thread Database，TDB）记录线程的所有相关数据，包括线程局部储存空间（Thread Local Storage，TLS）、消息队列、handle 表格、地址空间（Memory Context）等等等。

最初，进程是以一个线程（称为主线程，primary thread）做为开始。如果需要，进程可以产生更多的线程（利用 CreateThread），让 CPU 在同一时间执行不同段落的代码。当然，我们都知道，在只有一颗 CPU 的情况下，不可能真正有多任务的情况发生，「多个线程同时工作」的幻觉主要是靠排程器来完成 -- 它以一个硬件定时器和一组复杂的游戏规则，在不同的线程之间做快速切换动作。以Windows 95 和Windows NT而言，在非特殊的情况下，每个线程被CPU照顾的时间（所谓的timeslice）是20个milliseconds。

如果你有一部多 CPU 计算机，又使用一套支持多 CPU 的操作系统（如 Windows NT），那么一个 CPU 就可以分配到一个线程，真正做到实实在在的多任务。这种操作系统特性称为 symmetric multiprocessing（SMP）。Windows 95 没有SMP性质，所以即使在多CPU计算机上跑，也无法发挥其应有的高效能。

图 14-1 表现出一个进程（PDB）如何透过「MODREF 串列」连接到其所使用的所有模组。图 14-2 表现出一个模块数据结构（MDB）的细部内容，最后的 DataDirectory[16] 记录着16个特定节区（sections）的地址，这些sections 包括程序代码、数据、资源。图14-3 表现出一个线程数据结构（PDB）的细部内容。

当Windows加载器将程序加载内存中，KERNEL32 挖出一些内存，构造出一个PDB、一个TDB、一个以上的MDBs（视此程序使用到多少DLL 而定）。针对TDB，操作系统又要产生出memory context（就是在操作系统书籍中提到的那些所谓 page tables）、消息队列、handle 表格、环境数据结构（EDB）...。当这些系统内部数据结构都构造完毕，指令指位器（Instruction Pointer）移到程序的进入点，才开始程序的执行。

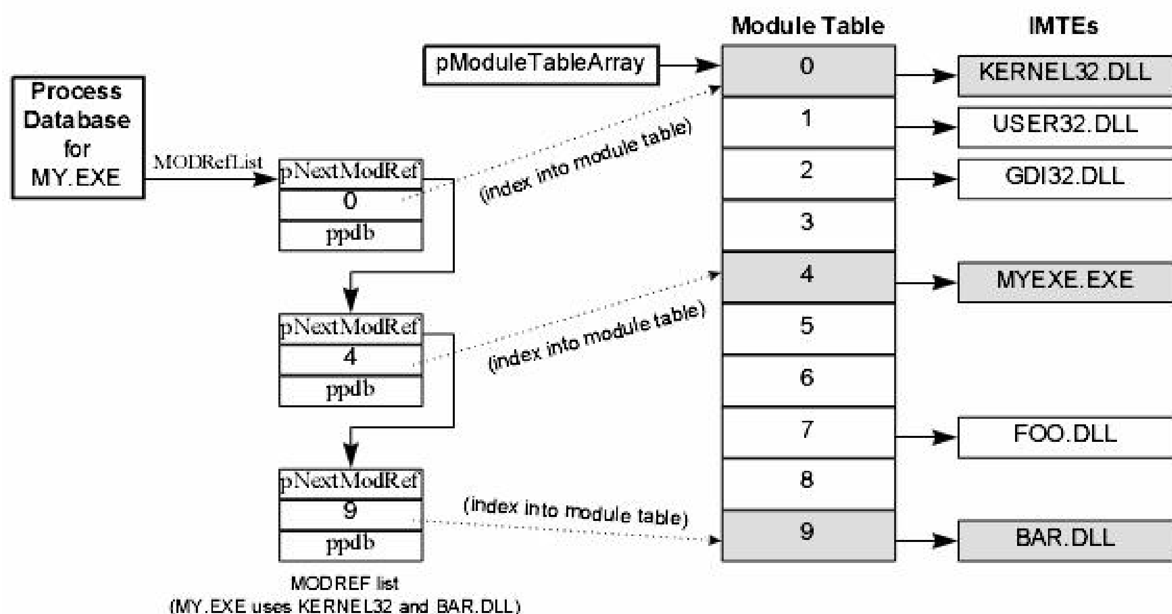


图14-1 进程（PDB）透过「MODREF串列」连接到其所使用的所有模块

线程优先权（Priority）

我想我们现在已经能够用很具体的形象去看所谓的进程、模块、线程了。「执行事实」发生在线程身上，而不在进程身上。也就是说，CPU 排程单位是线程而非进程。排程器据以排序的，是每个线程的优先权。

优先权的设定分为两个阶段。我已经在第 1 章介绍过。线程的「父亲大人」（进程）拥有所谓的优先权等级（priority class，图 1-7），可以在 CreateProcess 的参数中设定。线程基本上继承自其「父亲大人」的优先权等级，然后再加上 CreateThread 参数中的微调差额（-2~+2）。获得的结果（图 1-8）便是线程的所谓base priority，范围从0~31，数值愈高优先权愈高。::SetThreadPriority 是调整优先权的工具，它所指定的也是微调差额（-2~+2）。

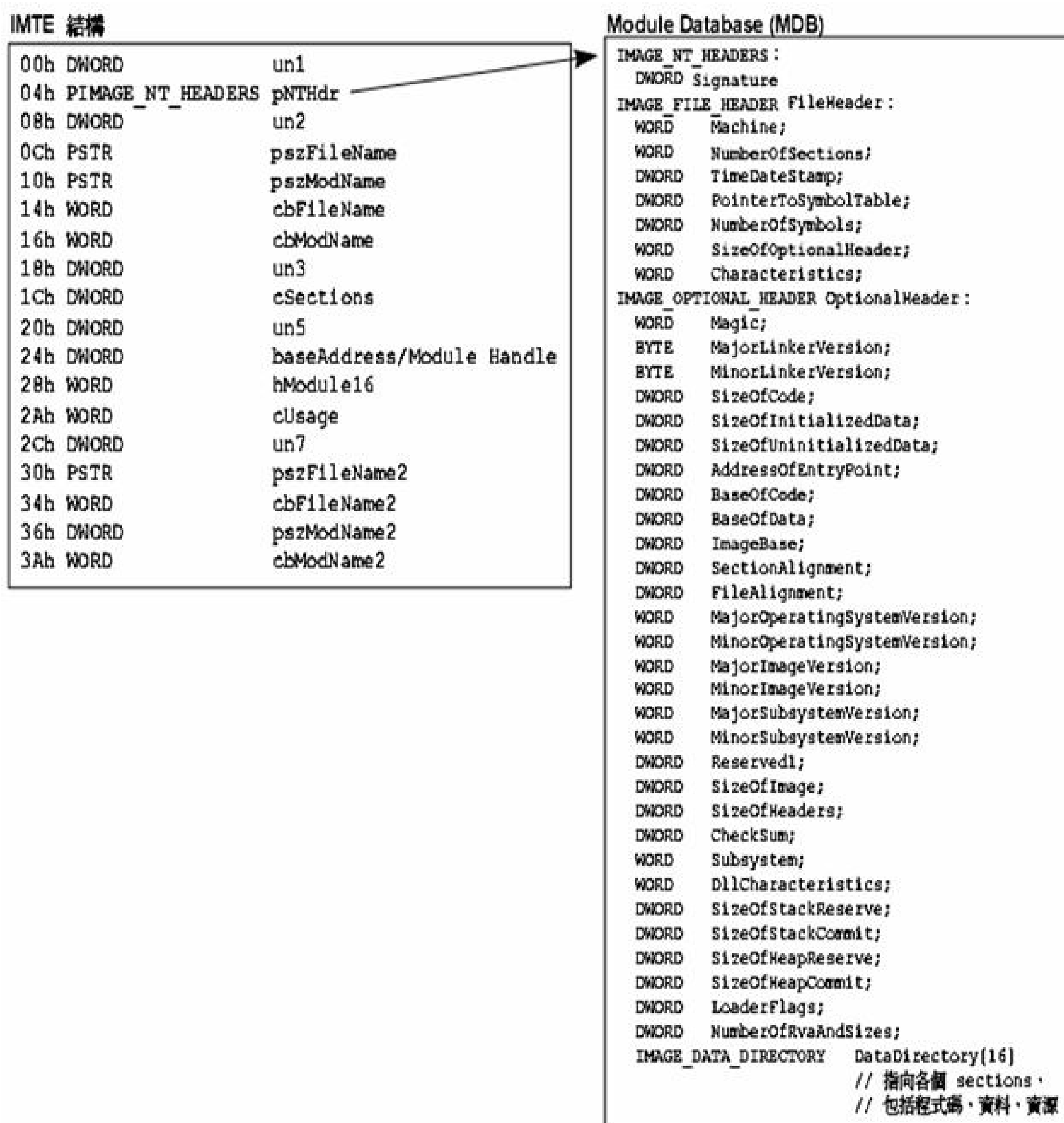


图14-2 模块数据结构 MDB 的细部内容（数据整理自Windows 95 System Programming SECRETS, Matt Pietrek, IDG Books）

线程排程（Scheduling）

排程器挑选「下一个获得 CPU 时间的线程」的唯一依据就是：线程优先权。如果所有等待被执行的线程中，有一个是优先权16，其它所有线程都是优先权15（或更低），那么优先权16者便是下一个夺标者。如果线程 A 和 B 同为优先权16，排程器会挑选等待比较久的那个（假设为线程 A）。当 A 的时间切片（timeslice）终了，如果 B 以外的其它线程的优先权仍维持在15（以下），线程 B 就会获得执行权。

「如果B以外的其它线程的优先权仍维持在 15（以下）...」，唔，这听起来彷佛优先权会变动似的。的确是。为了避免朱门酒肉臭、路有冻死骨的不公平情况发生，排程器会弹性调整线程优先权，以强化系统的反应能力，并且避免任何一个线程一直未能接受 CPU 的润泽。一般的线程优先权是 7，如果它被切换到前景，排程系统可能暂时地把它调升到8 或9 或更高。对于那些有着输入消息等待被处理的线程，排程系统也会暂时调高其优先权。

对于那些优先权本来就高的线程，也并不是有永久的保障权利。别忘了 Windows 毕竟是个消息驱动系统，如果某个线程调用 ::GetMessage 而其消息队列却是空的，这个线程便被冻结，直到再有消息进来为止。冻结的意思就是不管你的优先权有多高，暂时退出排班行列。线程也可能被以 ::SuspendThread 强制冻结住（::ResumeThread 可以解除冻结）。

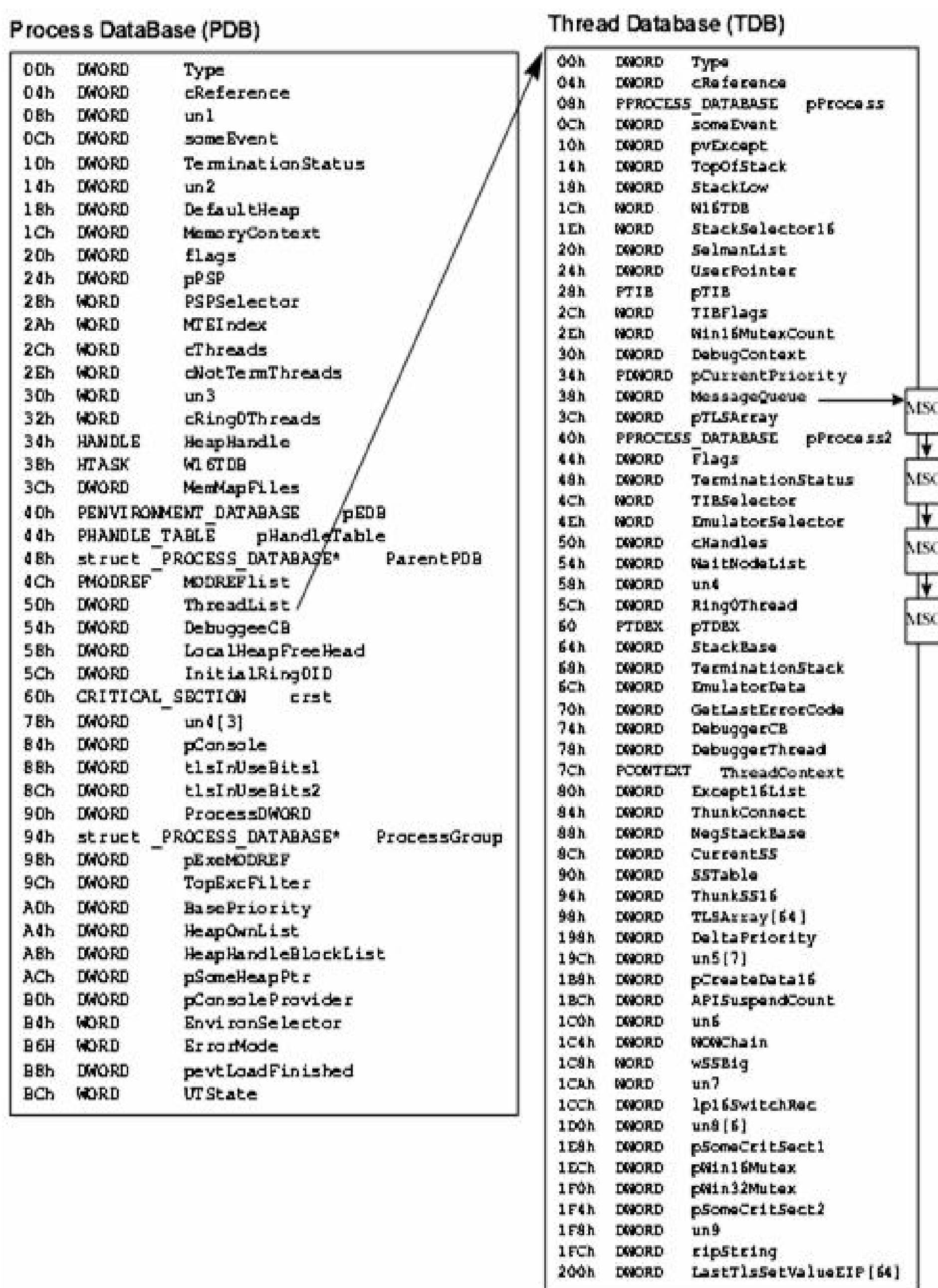


图14-3 线程数据结构（ PDB）的细部内容（数据整理自Windows 95 System Programming SECRETS,Matt Pietrek,IDG Books）

会被冻结，表示这个线程「要去抓取消息，而线程所附带的消息队列中却没有消息」。如果一个线程完全和 UI 无关呢？是否它就没有消息队列？倒不是，但它的程序代码中没有消息循环倒是事实。是的，这种线程称为 worker thread。正因它不可能会被冻结，所以它绝对不受 Win16Mutex 或其它因素而影响其强制性多任务性质，及其优先权。

Thread Context

Context 一词，我不知道有没有什么好译名，姑且就用原文吧。它的直接意思是「前后关系、脉络；环境、背景」。所以我们可以说Thread Context 是构成线程的「背景」。

那是指什么呢？狭义来讲是指一组缓存器值（包括指令指位器 IP）。因为线程常常会被暂停，被要求把CPU拥有权让出来，所以它必须将暂停之前一刻的状态统统记录下来，以备将来还可以恢复。

你可以在WINNT.H中找到一个CONTEXT资料结构， 它可以用来储存Thread Context 。::GetThreadContext 和::SetThreadContext 可 以 取 得 和 设 定 某个线程的context，因而改变该线程的状态。这已经是非常低阶的行为了。Matt Pietrek在其Windows 95 System Programming SECRETS 一书第10 章，写了一个Win32 API Spy程序，就充份运用了这两个函数。

我想我们在操作系统层面上的线程学理基础已经足够了，现在让我们看看比较实际一点的东西。

从程序设计层面看线程

书籍推荐：如果要从程序设计层面来了解线程，Jim Beveridge 和 Robert Wiener 合着的 Multithreading Applications in Win32 （Win32 多线程程序设计/侯俊杰译/碁峰出版）是很值得推荐的一份知识来源。这本书介绍线程的学理观念、程序方法、同步控制、资料一致性的保持、C runtime library 的多线程版本、C++ 的多线程程序方法、MFC 中的多线程程序方法、除错、进程通讯（IPC）、DLLs...，以及约 50 页的实际应用。

书籍推荐：Jeffrey Richter 的 Advanced Windows 在进程与线程的介绍上（第 2 章和第 3 章），也有非常好的表现。他的切入方式是详细而深入地叙述相关 Win32 API 的规格与用法。并举实例左证。

如何产生线程？我想各位都知道了，::CreateThread 可以办到。图 14-4 是与线程有关的 Win32 API。

```
与线程有关的Win32 API      功能
AttachThreadInput           将某个线程的输入导向另一个线程
CreateThread                产生一个线程
ExitThread                  结束一个线程
GetCurrentThread            取得目前线程的  handle
GetCurrentThreadId          取得目前线程的  ID
GetExitCodeThread           取得某一线程的结束代码（可用以决定线程是否已结束）
GetPriorityClass            取得某一进程的优先权等级
GetQueueStatus              传回某一线程的消息队列状态
GetThreadContext            取得某一线程的 context
GetThreadDesktop            取得某一线程的  desktop 对象
GetThreadPriority           取得某一线程的优先权
GetThreadSelectorEntry      除错器专用，传回指定之线程的某个selector 的LDT 记录项
ResumeThread                将某个冻结的线程恢复执行
SetPriorityClass            设定优先权等级
SetThreadPriority           设定线程的优先权
Sleep                       将某个线程暂时冻结。其它线程将获得执行权。
SuspendThread               冻结某个线程
TerminateThread             结束某个线程
TlsAlloc                    配置一个TLS（Thread Local Storage）
TlsFree                     释放一个TLS（Thread Local Storage）
TlsGetValue                 取得某个TLS（Thread Local Storage）的内容
TlsSetValue                 设定某个TLS（Thread Local Storage）的内容
WaitForInputIdle            等待，直到不再有输入消息进入某个线程中
```

图14-4 与线程有关的 Win32 API函数

注意，多线程并不能让程序执行得比较快（除非是在多 CPU 机器上，并且使用支持 symmetric multiprocessing 的操作系统），只是能够让程序比较「有反应」。试想某个程序在某个选单项目被按下后要做一个小时的运算工作，如果这份工作在主线程中做，而且没有利用 PeekMessage 的技巧时时观看消息队列的内容并处理之，那么这一个小时内这个程序的使用者接口可以说是被冻结住了，将毫无反应。但如果沉重的运算工作是由另一个线程来负责，使用者接口将依然灵活，不受影响。

Worker Threads 和 UI Threads

从Windows 操作系统的角度来看，线程就是线程，并未再有什么分类。但从MFC的角度看，则把线程划分为和使用者接口无关的worker threads，以及和使用者接口（UI）有关的UI threads。

基本上，当我们以 ::CreateThread 产生一个线程，并指定一个线程函数，它就是一个 worker thread，除非在它的生命中接触到了输入消息—— 这时候它应该有一个消息循环，以抓取消息，于是该线程摇身一变而为 UI thread。

注意，线程本来就带有消息队列，请看图 14-3 的 TDB 结构。而如果线程程序代码中带有一个消息循环，就称为 UI thread。

错误观念

我记得曾经在微软的技术文件中，也曾经在微软的范例程序中，看到他们鼓励这样的作法：为程序中的每一个窗口产生一个线程，负责窗口行为。这种错误的示范尤其存在于 MDI 程序中。是的，早期我也沾沾自喜地为 MDI 程序的每一个子窗口设计一个线程。基本上这是错误的行为，要付出昂贵的代价。因为子窗口一切换，上述作法会导至线程也切换，而这却要花费大量的系统资源。比较好的作法是把所有UI（User Interface）动作都集中在主线程中，其它的「纯种运算工作」才考虑交给worker threads去做。

正确态度

什么是使用多线程的好时机呢？如果你的程序有许多事要忙，但是你还要随时保持注意某些外部事件（可能来自硬件或来自使用者），这时就适合使用多线程来帮忙。

以通讯程序为例。你可以让主线程负责使用者接口，并保持中枢的地位。而以一个分离的线程处理通讯端口。

MFC 多线程程序设计

我已经在第 1 章以一个小节介绍了 Win32 多线程程序的写法，并给了一个小范例MltiThrd。这一节，我要介绍 MFC 多线程程序的写法。

探索 CWinThread

就像CWinApp 对象代表一个程序本身一样，CWinThread 对象代表一个线程本身。这个 MFC 类我们曾经看过，第 6 章讲「MFC 程序的生死因果」时，讲到「CWinApp::Run——程序生命的活水源头」，曾经追踪过CWinApp::Run 的源头 CWinThread::Run（里面有一个消息循环）。可见程序的「执行事实」系发生在 CWinThread 对象身上，而 CWinThread 对象必须要（必然会）产生一个线程。

我希望「CWinThread 对象必须要（必然会）产生一个线程」这句话不会引起你的误会，以为程序在application object（CWinApp 对象）的构造函数必然有个动作最终调用到 CreateThread 或 _beginthreadex。不，不是这样。想想看，当你的 Win32 程序执行起来，你的程序并没有调用 CreateProcess 为自己做出代表自己的那个进程，也没有调用 CreateThread 为自己做出代表自己的主线程（primary thread）的那个线程。为你的程序产生第一个进程和线程，是系统加载器以及核心模块（KERNEL32）合作的结果。

所以，再次循着第 6 章一步步剖析的步骤，MFC 程序的第一个动作是 CWinApp::CWinApp（比 WinMain 还早），在那里没有「产生线程」的动作，而是已经开始在收集线程的相关信息了：

```
// in MFC 4.2 APPCORE.CPP
CWinApp::CWinApp(LPCTSTR lpszAppName)
{
    ...
    // initialize CWinThread state
    AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
    AFX_MODULE_THREAD_STATE* pThreadState = pModuleState->m_thread;
    ASSERT(AfxGetThread() == NULL);
    pThreadState->m_pCurrentWinThread = this;
    ASSERT(AfxGetThread() == this);
    m_hThread = ::GetCurrentThread();
    m_nThreadID = ::GetCurrentThreadId();
    ...
}
```

虽然MFC程序只会有一个CWinApp对象，而CWinApp派生自CWinThread，但并不是说一个MFC程序只能有一个CWinThread 对象。每当你需要一个额外的线程，不应该在MFC程序中直接调用::CreateThread 或 _beginthreadex，应该先产生一个CWinThread 对象，再调用其成员函数 CreateThread 或全局函数 AfxBeginThread 将线程产生出来。当然，现在 你必然已经可以推测到，CWinThread::CreateThread或AfxBeginThread内部呼 叫了::CreateThread或_beginthreadex（事实上答案是_beginthreadex）。

这看起来颇有值得商议之处：为什么CWinThread 构造函数不帮我们调用 AfxBeginThread 呢？似乎CWinThread 为德不卒。

图 14-5 就是CWinThread 的相关原始代码。

```
#0001 // in MFC 4.2 THRDCORE.CPP
#0002 CWinThread::CWinThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam)
#0003 {
#0004         m_pfnThreadProc = pfnThreadProc;
#0005         m_pThreadParams = pParam;
#0006
#0007         CommonConstruct();
#0008 }
#0009
#0010 CWinThread::CWinThread()
#0011 {
#0012         m_pThreadParams = NULL;
#0013         m_pfnThreadProc = NULL;
#0014
#0015         CommonConstruct();
#0016 }
#0017
#0018 void CWinThread::CommonConstruct()
#0019 {
#0020         m_pMainWnd = NULL;
#0021         m_pActiveWnd = NULL;
#0022
#0023         // no HTHREAD until it is created
#0024         m_hThread = NULL;
#0025         m_nThreadID = 0;
#0026
#0027         // initialize message pump
#0028 #ifdef _DEBUG
#0029         m_nDisablePumpCount = 0;
#0030 #endif
#0031         m_msgCur.message = WM_NULL;
#0032         m_nMsgLast = WM_NULL;
#0033         ::GetCursorPos(&m_ptCursorLast);
#0034
#0035         // most threads are deleted when not needed
#0036         m_bAutoDelete = TRUE;
#0037
#0038         // initialize OLE state
#0039         m_pMessageFilter = NULL;
#0040         m_lpfnOleTermOrFreeLib = NULL;
#0041 }
#0042
#0043 CWinThread* AFXAPI AfxBeginThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam,
#0044         int nPriority, UINT nStackSize, DWORD dwCreateFlags,
#0045         LPSECURITY_ATTRIBUTES lpSecurityAttrs)
#0046 {
#0047     CWinThread* pThread = DEBUG_NEW CWinThread(pfnThreadProc, pParam);
#0048
#0049     if (!pThread->CreateThread(dwCreateFlags|CREATE_SUSPENDED, nStackSize,
#0050         lpSecurityAttrs))
#0051     {
#0052         pThread->Delete();
#0053         return NULL;
#0054     }
#0055     VERIFY(pThread->SetThreadPriority(nPriority));
#0056     if (!(dwCreateFlags & CREATE_SUSPENDED))
#0057         VERIFY(pThread->ResumeThread() != (DWORD)-1);
#0058
#0059     return pThread;
#0060 }
#0061
#0062 CWinThread* AFXAPI AfxBeginThread(CRuntimeClass* pThreadClass,
#0063         int nPriority, UINT nStackSize, DWORD dwCreateFlags,
```

```
#0064          LPSECURITY_ATTRIBUTES lpSecurityAttrs)
#0065  {
#0066          ASSERT(pThreadClass != NULL);
#0067          ASSERT(pThreadClass->IsDerivedFrom(RUNTIME_CLASS(CWinThread)));
#0068
#0069          CWinThread* pThread = (CWinThread*)pThreadClass->CreateObject();
#0070
#0071          pThread->m_pThreadParams = NULL;
#0072          if (!pThread->CreateThread(dwCreateFlags|CREATE_SUSPENDED, nStackSize,
#0073                lpSecurityAttrs))
#0074          {
#0075                pThread->Delete();
#0076                return NULL;
#0077          }
#0078          VERIFY(pThread->SetThreadPriority(nPriority));
#0079          if (!(dwCreateFlags & CREATE_SUSPENDED))
#0080                VERIFY(pThread->ResumeThread() != (DWORD)-1);

#0081
#0082          return pThread;
#0083  }
#0084
#0085  BOOL CWinThread::CreateThread(DWORD dwCreateFlags, UINT nStackSize,
#0086          LPSECURITY_ATTRIBUTES lpSecurityAttrs)
#0087  {
#0088      // setup startup structure for thread initialization
#0089      _AFX_THREAD_STARTUP startup; memset(&startup, 0, sizeof(startup));
#0090      startup.pThreadState = AfxGetThreadState();
#0091      startup.pThread = this;
#0092      startup.hEvent = ::CreateEvent(NULL, TRUE, FALSE, NULL);

#0093      startup.hEvent2 = ::CreateEvent(NULL, TRUE, FALSE, NULL);
#0094      startup.dwCreateFlags = dwCreateFlags;
#0095      ...
#0096      // create the thread (it may or may not start to run)
#0097      m_hThread = (HANDLE)_beginthreadex(lpSecurityAttrs, nStackSize,
#0098        &_AfxThreadEntry, &startup, dwCreateFlags | CREATE_SUSPENDED, (UINT*)&m_nThreadID);
#0099      ...
#0100  }
```

图14-5 CwinThread的相关原始代码

产生线程，为什么不直接用::CreateThread 或_beginthreadex？为什 么要透过CWinThread对象 ？我想你可以轻易从MFC原始代码中看出，因为CWinThread::CreateThread 和 AfxBeginThread 不只是::CreateThread 的一层包装，更做了一些application framework 所需的内部数据初始化工作，并确保使用正确的C runtime library 版本。原始代码中有：

```
#ifndef _MT
... // 做些设定工作，不产生线程，回返。
#else
... // 真正产生线程，回返。
#endif //!_MT)
```

的动作，只是被我删去未列出而已。

接下来我要把worker thread和UI thread 的产生步骤做个整理。它们都需要调用 AfxBeginThread 以产生一个CWinThread 对象，但如果要产生一个 UI thread，你还必须先定义一个 CWinThread 派生类。

产生一个Worker Thread

Worker thread 不牵扯使用者接口。你应该为它准备一个线程函数，然后调用 AfxBeginThread：

```
CWinThread* pThread = AfxBeginThread(ThreadFunc, &Param);
...
UINT ThreadFunc (LPVOID pParam)
{
    ...
}
```

AfxBeginThread 事实上一共可以接受六个参数，分别是：

```
CWinThread* AFXAPI AfxBeginThread(AFX_THREADPROC pfnThreadProc,
LPVOID pParam,
int nPriority = THREAD_PRIORITY_NORMAL,
UINT nStackSize = 0,
DWORD dwCreateFlags = 0,
LPSECURITY_ATTRIBUTES lpSecurityAttrs= NULL);
```

参数一pfnThreadProc 表示线程函数。参数二pParam 表示要传给线程函数的参数。参数三 nPriority表示优先权的微调值，预设为 THREAD_PRIORITY_NORMAL，也就是没有微调。参数四 nStackSize 表示堆栈的大小，默认值 0 则表示堆栈最大容量为1MB。参数五 dwCreateFlags 如果为默认值 0，就表示线程产生后立刻开始执行；如果其值为 CREATE_SUSPENDED，就表示线程产生后先暂停执行。之后你可以使用 CWinThread::ResumeThread 重新执行它。参数六 lpSecurityAttrs 代表新线程的安全防护属性。默认值 NULL 表示此一属性与其产生者（也是个线程）的属性相同。

在这里我们遭遇到一个困扰。线程函数是由系统调用的，也就是个 callback 函数，不容许有 this 指针参数。所以任何一般的 C++ 类成员函数都不能够拿来当做线程函数。它必须是个全局函数，或是个 C++ 类的 static 成员函数。其原因我已经在第 6 章的「Callback 函数」一节中描述过了，而采用全局函数或是C++ static 成员函数，其间的优劣因素我也已经在该节讨论过。

线程函数的类型AFX_THREADPROC定义于AFXWIN.H 之中：

```
// in AFXWIN.H
typedef UINT (AFX_CDECL *AFX_THREADPROC)(LPVOID);
```

所以你应该把本身的线程函数声明如下（其中的 pParam 是个指针，在实用上可以指向程序员自定的数据结构）：

```
UINT ThreadFunc (LPVOID pParam);
```

否则，编译时会获得这样的错误消息：

```
error C2665: 'AfxBeginThread' : none of the 2 overloads can convert
parameter 1 from type 'void (unsigned long *)'
```

有时候我们会让不同的线程使用相同的线程函数，这时候你就得特别注意到线程函数使用全局变量或静态变量时，数据共享所引发的严重性（有好有坏）。至于放置在堆栈中的变量或对象，都不会有问题，因为每一个线程自有一个堆栈。

产生一个UI Thread

UI thread 可不能够光由一个线程函数来代表，因为它要处理消息，它需要一个消息循环。好得很，CWinThread::Run 里头就有一个消息循环。所以，我们应该先从CWinThread派生一个自己的类，再调用 AfxBeginThread 产生一个 CWinThread 对象：

```
class CMyThread : public CWinThread
{
    DECLARE_DYNCREATE(CMyThread)
    public:
    void BOOL InitInstance();
};
IMPLEMENT_DYNCREATE(CMyThread, CWinThread)
BOOL CMyThread::InitInstance()
{
    ...
}
CWinThread *pThread = AfxBeginThread(RUNTIME_CLASS(CMyThread));
```

我想你对RUNTIME_CLASS 宏已经不陌生了，第 3 章和第 8 章都有这个宏的原始代码展现以及意义解释。AfxBeginThread 是上一小节同名函数的一个 overloaded 函数，一共可以接受五个参数，分别是：

```
CWinThread* AFXAPI AfxBeginThread(CRuntimeClass* pThreadClass,
int nPriority = THREAD_PRIORITY_NORMAL,
UINT nStackSize = 0,
DWORD dwCreateFlags = 0,
LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);
```

最后四个参数的意义和默认值比上一节同名函数相同，但是少接受一个 LPVOID pParam 参数。

你可以在AFXWIN.H 中找到CWinThread的定义：

```
class CWinThread : public CCmdTarget
{
    DECLARE_DYNAMIC(CWinThread)
    BOOL CreateThread(DWORD dwCreateFlags = 0, UINT nStackSize = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);
    ...
    int GetThreadPriority();
    BOOL SetThreadPriority(int nPriority);
    DWORD SuspendThread();
    DWORD ResumeThread();
    BOOL PostThreadMessage(UINT message, WPARAM wParam, LPARAM lParam);
    ...
};
```

其中有许多成员函数和图14-4中的Win32 API函数有关。在CWinThread的成员函数中，有五个函数只是非常单纯的 Win32 API 的包装而已，它们被定义于AFXWIN2.INL文件中：

```
// in AFXWIN2.INL
// CWinThread
_AFXWIN_INLINE BOOL CWinThread::SetThreadPriority(int nPriority)
{ ASSERT(m_hThread!=NULL);return ::SetThreadPriority(m_hThread, nPriority); }
_AFXWIN_INLINE int CWinThread::GetThreadPriority()
{ ASSERT(m_hThread != NULL); return ::GetThreadPriority(m_hThread); }
_AFXWIN_INLINE DWORD CWinThread::ResumeThread()
{ ASSERT(m_hThread != NULL); return ::ResumeThread(m_hThread); }
_AFXWIN_INLINE DWORD CWinThread::SuspendThread()
{ ASSERT(m_hThread != NULL); return ::SuspendThread(m_hThread); }
_AFXWIN_INLINE BOOL CWinThread::PostThreadMessage(UINT message, WPARAM wParam, LPARAM lPa
{ ASSERT(m_hThread != NULL); return ::PostThreadMessage(m_nThreadID, message, wParam, lPa
```

线程的结束

既然worker thread 的生命就是线程函数本身，函数一旦 return，线程也就结束了，自然得很。或者线程函数也可以调用 AfxEndThread，结束一个线程。

UI 线程因为有消息循环的关系，必须在消息队列中放一个 WM_QUIT，才能结束线程。放置的方式和一般Win32程序一样，调用::PostQuitMessage 即可办到。亦或者，在线程的任何一个函数中调用AfxEndThread，也可以结束线程。

AfxEndThread 其实也是个外包装，其内部调用 _endthreadex，这个动作才真正把线程结束掉。

别忘了，不论worker thread或UI thread，都需要一个CWinThread 对象，当线程结束，记得把该对象释放掉（利用 delete）。

线程与同步控制

看起来线程的诞生与结束，以及对它的优先权设定、冻结、重新启动，都很容易。但是我必须警告你，多线程程序的设计成功关键并不在此。如果你的每一个线程都非常独立，彼此没有干联，也就罢了。但如果许多个线程互有关联呢？有经验的人说多线程程序设计有多复杂

多困难，他们说的并不是线程本身，而是指线程与线程之间的同步控制。

原因在于，没有人能够预期线程的被执行。在一个合作型多任务系统中（例如 Windows 3.x），操作系统必须得到程序的允许才能够改变线程。但是在强制性多任务系统中（如 Win95或WinNT），控制权被排程器强制移转，也因此两个线程之间的执行次序变得不可预期。这不可预期性造成了所谓的 race conditions。

假设你正在一个文件服务器中编辑一串电话号代码。文件打开来内容如下：

```
Charley     572-7993
Graffie     573-3976
Dennis      571-4219
```

现在你打算为Sue加上一笔新数据。正当你输入 Sue电话号代码的时候，另一个人也打开文件并输入另一笔有关于 Jason 的数据。最后你们两人也都做了存文件动作。谁的数据会留下来？答案是比较晚存盘的那个人，而前一个人的输入会被覆盖掉。这两个人面临的就是 race condition。

再举一个例子。你的程序产生两个线程，A和B。线程B的任务是设定全局变量X。线程A则要去读取X。假设线程B先完成其工作，设定了X，然后线程A才执行，读取X，这是一种好的情况，如图 14-6a。但如果线程A先执行起来并读取全局变量X，它会读到一个不适当的值，因为线程B还没有完成其工作并设定适当的X。如图14-6b。这也是race condition。

另一种线程所造成的可能问题是：死结（deadlock）。图14-7可以说明这种情况。

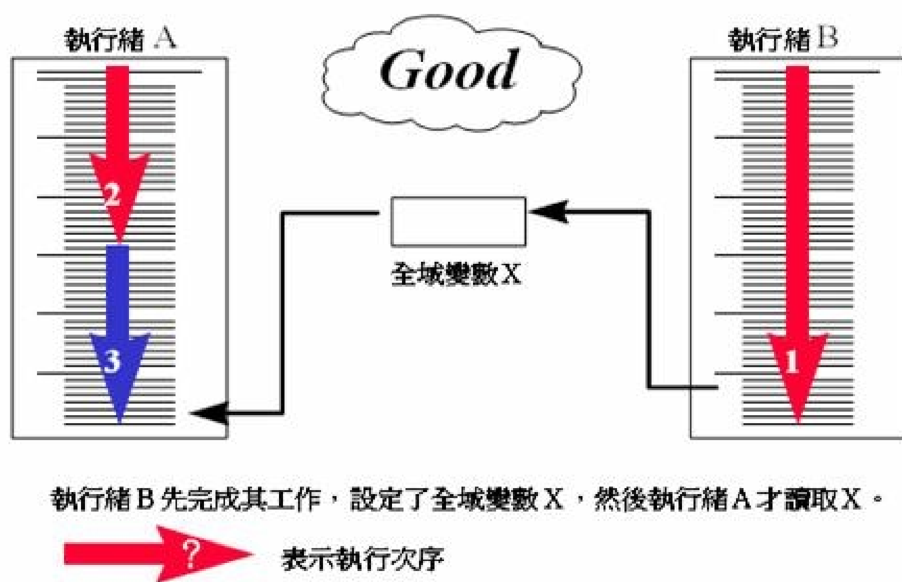


图14-6a race condition（good）

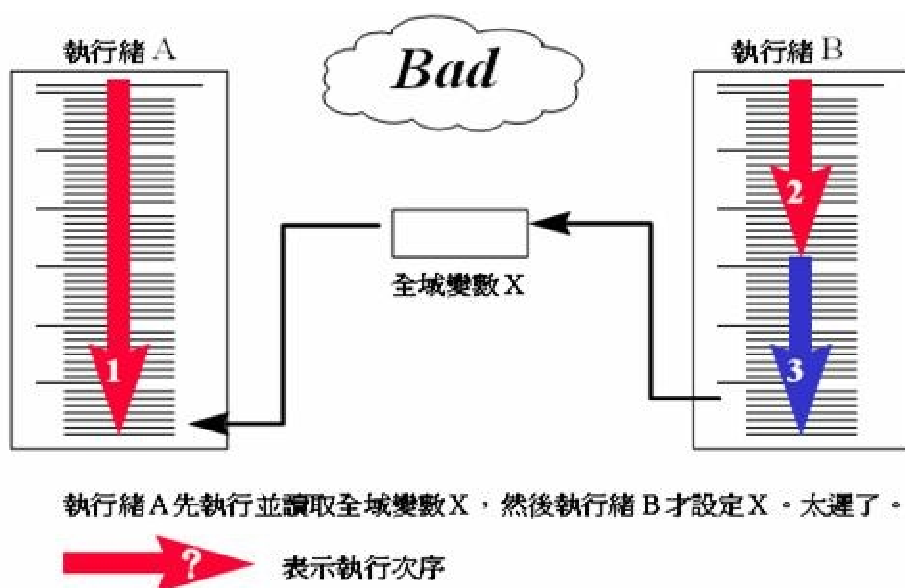


图14-6b race condition（bad）

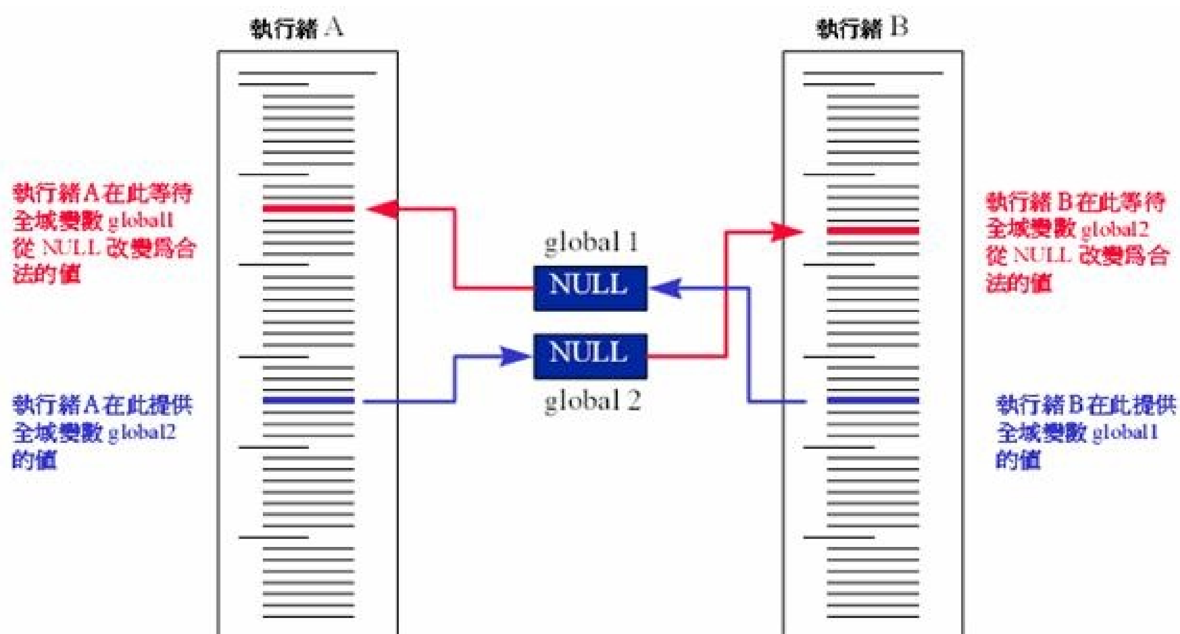


图14-7 死结（deadlock）

要解决这些问题，必须有办法协调各个线程的执行次序，让某个线程等待某个线程。Windows 系统提供四种同步化机制，帮助程序进行这种工作：

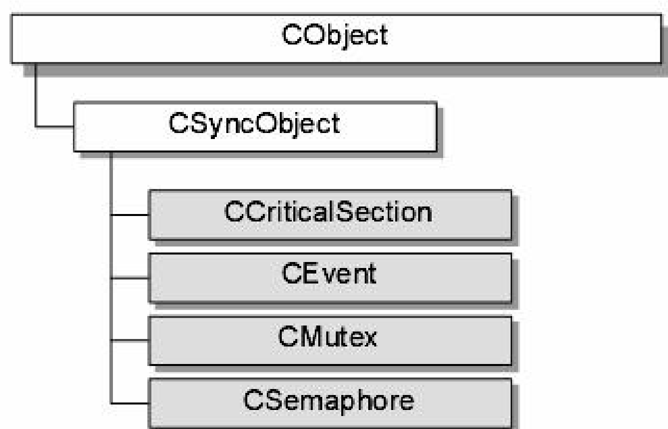
1. Critical Section（关键局部）

2. Semaphore（号志）

3. Event（事件）

4. Mutex（Mutual Exclusive，互斥器）

MFC 也提供了四个对应的类：



MFC 多线程程序实例

我将在此示范如何把第 1 章最后的一个 Win32 多线程程序 MltiThrd 改装为 MFC 程序。我只示范主架构（与 CWinThread、AfxBeginThread、ThreadFunc 有关的部份），程序绘图部份留给您做练习。

首先我利用MFC AppWizard 产生一个Mltithrd 项目，放在书附盘片的 Mltithrd.14 子目录中，并接受 MFC AppWizard 的所有预设选项。

接下来我在resource.h 中加上一些定义，做为线程函数的参数，以便在绘图时能够把代表各线程的各个长方形涂上不同的颜色：

```
#define HIGHEST_THREAD      0x00
#define ABOVE_AVE_THREAD    0x3F
#define NORMAL_THREAD       0x7F
#define BELOW_AVE_THREAD    0xBF
#define LOWEST_THREAD       0xFF
```

然后我在Mltithrd.cpp 中加上一些全局变量（你也可以把它们放在 CMltithrdApp 之中。我只是为了图个方便）：

```
CMltithrdApp theApp;
CWinThread* _pThread[5];
DWORD _ThreadArg[5] = { HIGHEST_THREAD,    // 0x00
                        ABOVE_AVE_THREAD,  // 0x3F
                        NORMAL_THREAD,     // 0x7F
                        BELOW_AVE_THREAD,  // 0xBF
                        LOWEST_THREAD      // 0xFF
                      };    // 用來調整四方形顏色
```

然后在 CMltithrdApp::InitInstance 函数最后面加上一些代码：

```
// create 5 threads and suspend them
int i;
for (i= 0; i< 5; i++)
{
    _pThread[i] = AfxBeginThread(CMltithrdView::ThreadFunc,
                                 &_ThreadArg[i],
                                 THREAD_PRIORITY_NORMAL,
                                 0,
                                 CREATE_SUSPENDED,
                                 NULL);
}
```

```
// setup relative priority of threads
_pThread[0]->SetThreadPriority(THREAD_PRIORITY_HIGHEST);
_pThread[1]->SetThreadPriority(THREAD_PRIORITY_ABOVE_NORMAL);
_pThread[2]->SetThreadPriority(THREAD_PRIORITY_NORMAL);
_pThread[3]->SetThreadPriority(THREAD_PRIORITY_BELOW_NORMAL);
_pThread[4]->SetThreadPriority(THREAD_PRIORITY_LOWEST);
```

这样一来我就完成了五个 worker threads 的产生，并且将其优先权做了 -2~+2 范围之间的微调。

接下来我应该设计线程函数。就如我在第 1 章已经说过，这个函数的五个线程可以使用同一个线程函数。本例中是设计为全局函数好呢？还是 static 成员函数好？如果是后者，应该成为哪一个类的成员函数好？

为了「要在线程函数做窗口绘图动作」的考虑，我把线程函数设计为 CMltithrdView的一个 static 成员函数，并遵循应有的函数类型：

```
// in MltithrdView.h
class CMltithrdView : public CView
{
...
public:
       CMltithrdDoc* GetDocument();
       static UINT ThreadFunc(LPVOID);
...
};

// in MltithrdView.cpp
UINT CMltithrdView::ThreadFunc(LPVOID ThreadArg)
{
    DWORD dwArg = *(DWORD*)ThreadArg;

    // ... 在這裡做如同第 1 章的 MltitThrd 一樣的繪圖動作
    return 0;
}
```

好，到此为止，编译链接，获得的程序将在执行后产生五个线程，并全部冻结。以 Process Viewer（Visual C++ 5.0 所附工具）观察之，证明它的确有六个线程（包括一个主线程以及我们所产生的另五个线程）：

Process Viewer Application

File Process Help

Process	PID	Base Priority	Num. Threads	Type	Full Path
MLTITHRD.EXE	FFC23175	8 (Normal)	6	32-Bit	H:\U004\PROG\MLTITHRD.14\DEBUG\MLTI...
PVIEW95.EXE	FFC3670D	8 (Normal)	1	32-Bit	E:\DEVSTUDIO\VC\BIN\WIN95\PVIEW95.EXE
WINOLDAP	FFC2A631	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\WINOA386.MOD
WINOLDAP	FFC2D4CD	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\WINOA386.MOD
WINWORD.EXE	FFC2E431	8 (Normal)	1	32-Bit	D:\MSOFFICE\WINWORD\WINWORD.EXE
MSDEV.EXE	FFFD770D	8 (Normal)	7	32-Bit	E:\DEVSTUDIO\SHAREDIDE\BIN\MSDEV.EXE
PPSHELL.EXE	FFFC8E41	8 (Normal)	1	32-Bit	C:\PPENSB\WIN32\PPSHELL.EXE
ETENSRV	FFFD7CF1	8 (Normal)	1	16-Bit	D:\Program Files\ET2K\BOX95\ETENSRV.EXE
SAGE.EXE	FFFCFA3D	8 (Normal)	2	32-Bit	D:\WIN95\SYSTEM\SAGE.EXE
SYSTRAY.EXE	FFFCE72D	8 (Normal)	1	32-Bit	D:\WIN95\SYSTEM\SYSTRAY.EXE
INTERNAT.EXE	FFFCD6BD	8 (Normal)	1	32-Bit	D:\WIN95\SYSTEM\INTERNAT.EXE
EXPLORER.EXE	FFFC0CF1	8 (Normal)	3	32-Bit	D:\WIN95\EXPLORER.EXE
MMTASK	FFFC60B5	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\mmtask.tsk

TID	Owning PID	Thread Priority
FFC041C9	FFC23175	6 (Lowest)
FFC3BE51	FFC23175	7 (Below Normal)
FFC3BB39	FFC23175	8 (Normal)
FFC3B981	FFC23175	9 (Above Norm...
FFC3B669	FFC23175	10 (Highest)
FFC23535	FFC23175	8 (Normal)

接下来，留给你的作业是：

1. 利用资源编辑器为程序加上各选单项目，如图 1-9。

2. 设计上述选单项目的命令处理例程。

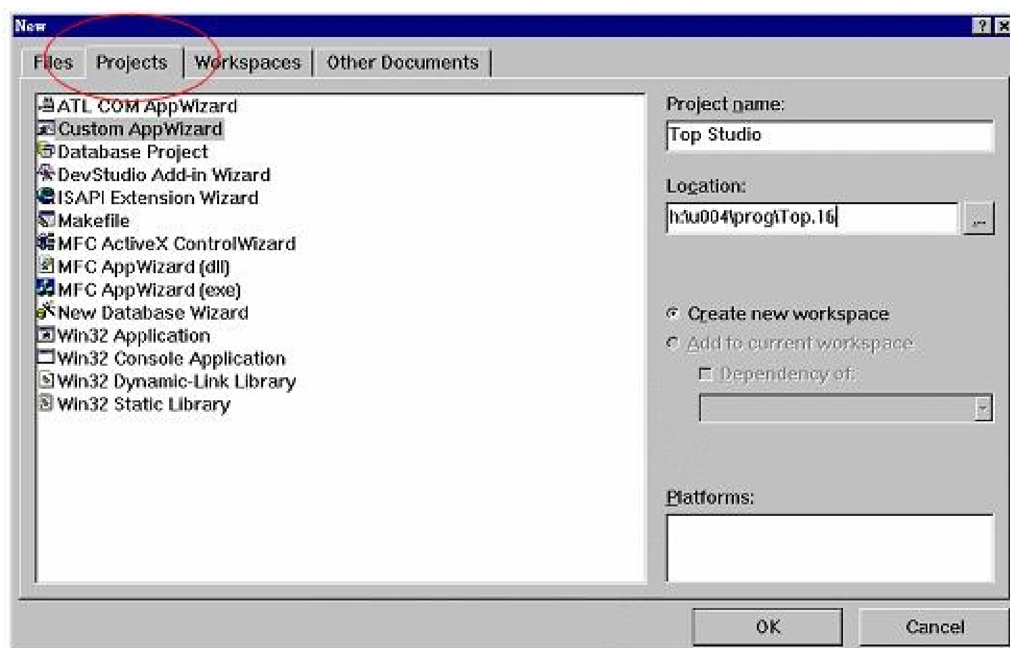
3. 在线程函数ThreadFunc 内加上计算与绘图能力。并判断使用者选择何种延迟方式，做出适当反应。

第15章 定制一个AppWizard

我们的Scribble 程序一路走来，大家可还记得它一开始并不是平地而起，而是由AppWizard以「程序代码产生器」的身份，自动为我们做出一个我所谓的「骨干程序」来？

Developer's Studio 提供了一个开放的 AppWizard 接口。现在，我们可以轻易地扩充AppWizard：从小规模的扩充，到几乎改头换面成为一种全新类型的程序代码产生器。

Developer's Studio 提供了许多种不同的项目类型，供你选择。当你选按 Visual C++ 5.0 整合环境中的【File/New】命令项，并选择【Projects】附页，便得到这样的对话窗画面：



除了上述这些内建的程序类型，它还可以显示出任何自定程序类型（custom types）。Developer's Studio（整合环境）和 AppWizard 之间的接口借着一组类和一些组件表现出来，使我们能够轻易订制合乎自己需求的 AppWizard。制造出来的所谓custom AppWizard（一个扩展名为 .AWX 的动态链接函数库，注），必须被放置于磁盘目录 \DevStudio\SharedIDE\Template 中，才能发挥效用。Developers Studio 和 AppWizard 和 AWX 之间的基本架构如图 15-1。

注：我以 DUMPBIN（Visual C++ 附的一个观察文件类型的工具）观察 .AWX 档，得到结果如下：

```
E:\DevStudio\SharedIDE\BIN\IDE>dumpbin addinwz.awx
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997\. All rights reserved.
Dump of file addinwz.awx
File Type: DLL <--- 这证明 .AWX 的确是个动态链接函数库。
Summary
1000 .data
1000 .reloc
3A000 .rsrc
3000 .text
```

事实上AWX（Application Wizard eXtension）就是一个32位元的MFC extension DLL。

是不是Visual C++ 系统中早已存在有一些 .AWX 檔了呢？当然，它们是：

```
Directory of E:\DevStudio\SharedIDE\BIN\IDE
ADDINWZ  AWX      255,872  03-29-97  16:43 ADDINWZ.AWX
ATLWIZ   AWX      113,456  03-29-97  16:43 ATLWIZ.AWX
CUSTMWZ  AWX      278,528  03-29-97  16:43 CUSTMWZ.AWX
INETAWZ  AWX      91,408  03-29-97  16:43 INETAWZ.AWX
MFCTLWZ  AWX      146,272  03-29-97  16:43 MFCTLWZ.AWX
5 file(s)          885,536 bytes
```

请放心，你只能够扩充（新增）项目类型，不会一不小心取代了某一个原已存在的项目类型。

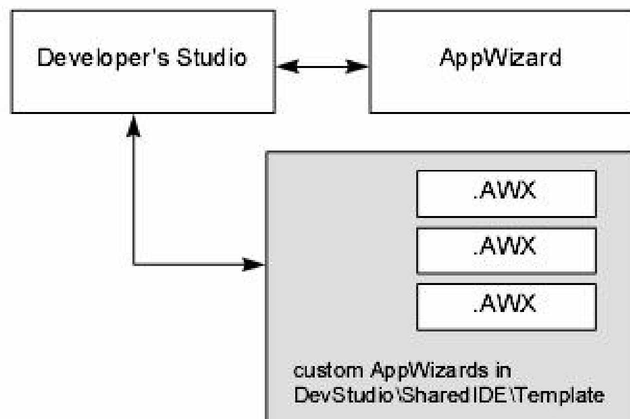


图15-1 Developers Studio和AppWizard 和*.AWX 之间的基本架构

到底Wizard是什么？

所谓Wizard，就是一个扩展名为.AWX 的动态链接函数库。Visual C++ 的 "project manager" 会检查整合环境中的Template 子目录（\DevStudio\SharedIDE\Template），然后显示其图标于【New Project】对话窗中，供使用者选择。

Wizard 本身是一个所谓的「template 直译器」。这里所谓的"template" 是一些文字文件，内有许多特殊符号（也就是本章稍后要介绍的macros 和 directives）。Wizard 读取这些template，对于正常文字，就以正常的output stream 输出到另一个文件中；对于特殊符号或保留字，就解析它们然后再把结果以一般的 output stream 输出到文件中。Wizard 所显示给

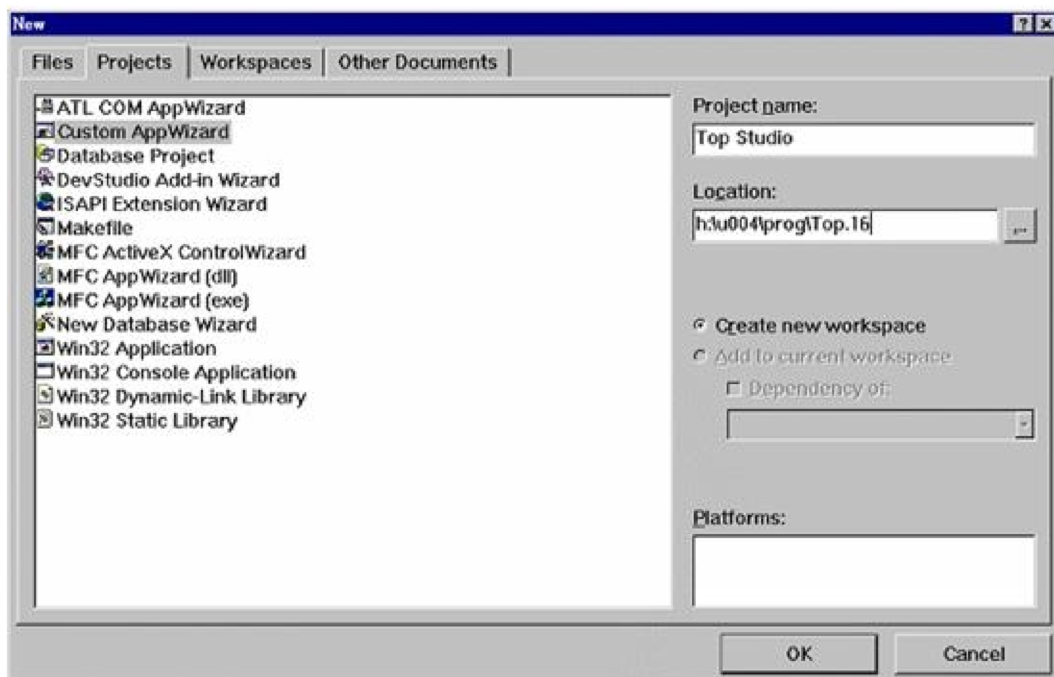
使用者看的「步骤对话窗」可以接受使用者的指定项目或文字输出，于是会影响 template 中的特殊符号的内容或解析，连带也就影响了Wizard 的stream 输出。这些stream 输出，最后就成为你的项目的源文件。

Custom AppWizard 的基本操作

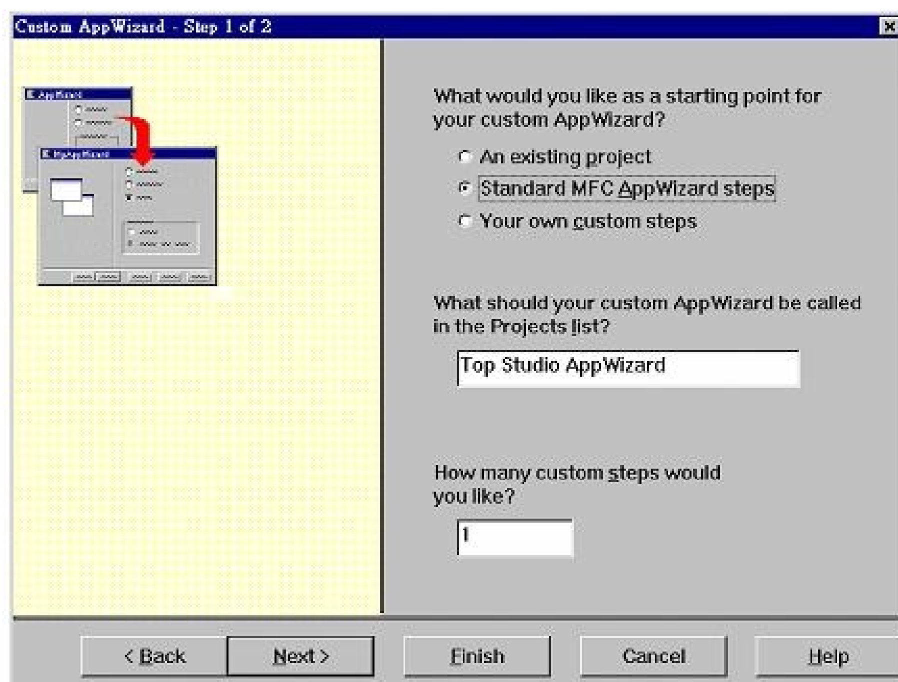
Developers Studio提供了一个让我们制作custom AppWizard的Wizard，就叫作 Custom AppWizard。让我们先实地操作一下这个工具，再来谈程序技术问题。

注意：以下我以 Custom AppWizard 表示 Visual C++ 所附的工具，custom AppWizard 表示我们希望做出来的「订制型 AppWizard」。

选按【File/New】，在对话窗中选择 Custom AppWizard，然后在右边填写你的项目名称：



按下【OK】钮，进入步骤一画面：



你可以选择三种可能的扩充型式：

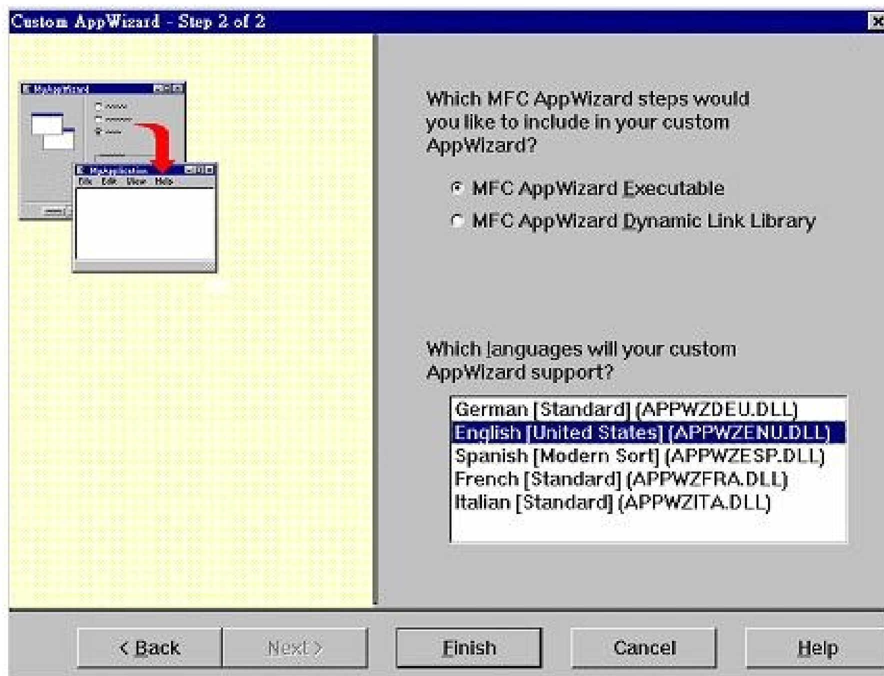
1. An existing project：根据一个原已存在的项目文件（*.dsp）来产生一个custom AppWizard。

2. Standard MFC AppWizard steps：根据某个原有的AppWizard，为它加上额外的几个步骤，成为一个新的 custom AppWizard。这是一般最被接受的一种方式。

3. Your own custom steps：有全新的步骤和全新的对话窗画面。这当然是最大弹性的展现啦，并同时也是最困难的一种作法，因为你要自行负责所有的工作。哪些工作呢？稍后我有一个例子使用第二种型式，将介绍所谓的macros和directievs，你可以从中推而想之这第三种型式的繁重负担。

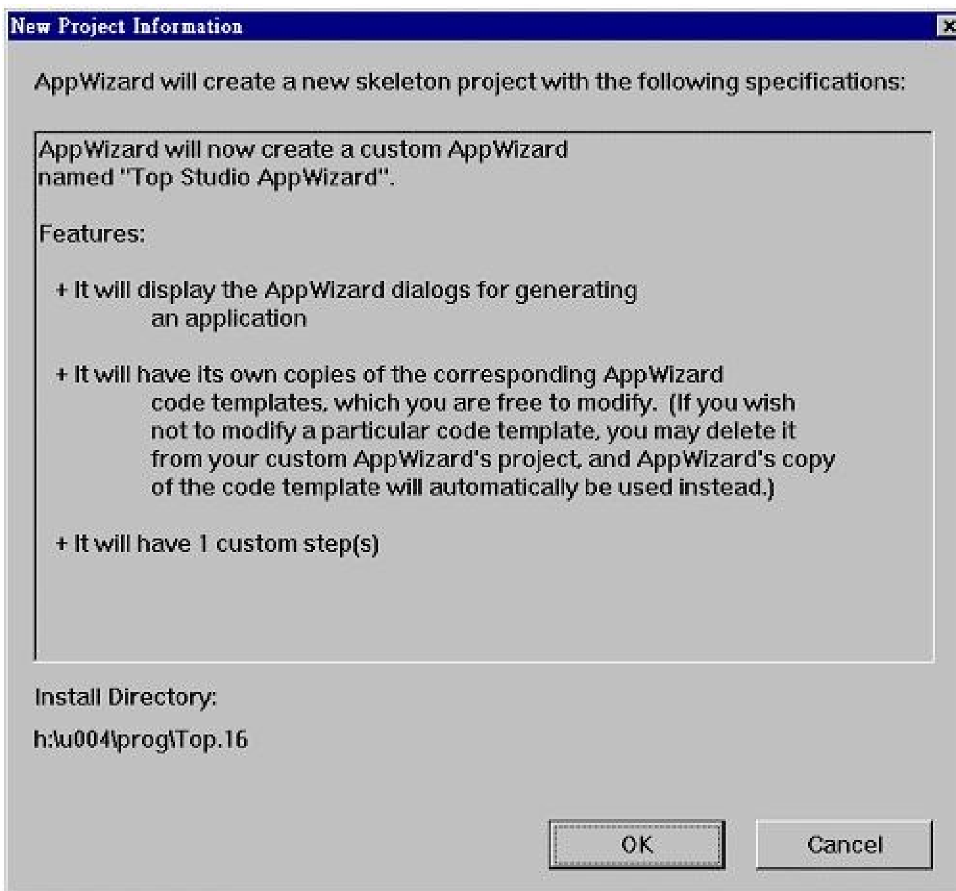
我的目的是做出一个属于我个人研究室（"Top Studio"）专用的custom AppWizard，以原本的MFC AppWizard（exe）为基础（有六个步骤），再加上一页（一个步骤），让程序员填入姓名、简易说明，然后Top Studio AppWizard 能够把这些数据加到每一个原始代码文件最前端。所以，我应该选择上述三种情况的第二种：Standard MFC AppWizard steps，并在上图下方选择欲增加的步骤数量。本例为1。

接下来按【Next】进入 Custom AppWizard 的第二页：



既然刚刚选择的是 Standard MFC AppWizard steps，这第二页便问你要制造出MFC Exe或MFC Dll。我选择MFC Exe。并在对话窗下方选择使用的文字：英文。很可惜目前这里没有中文可供选择。

这样就完成了订制的程序。按下【Finish】钮，你获得一张清单：



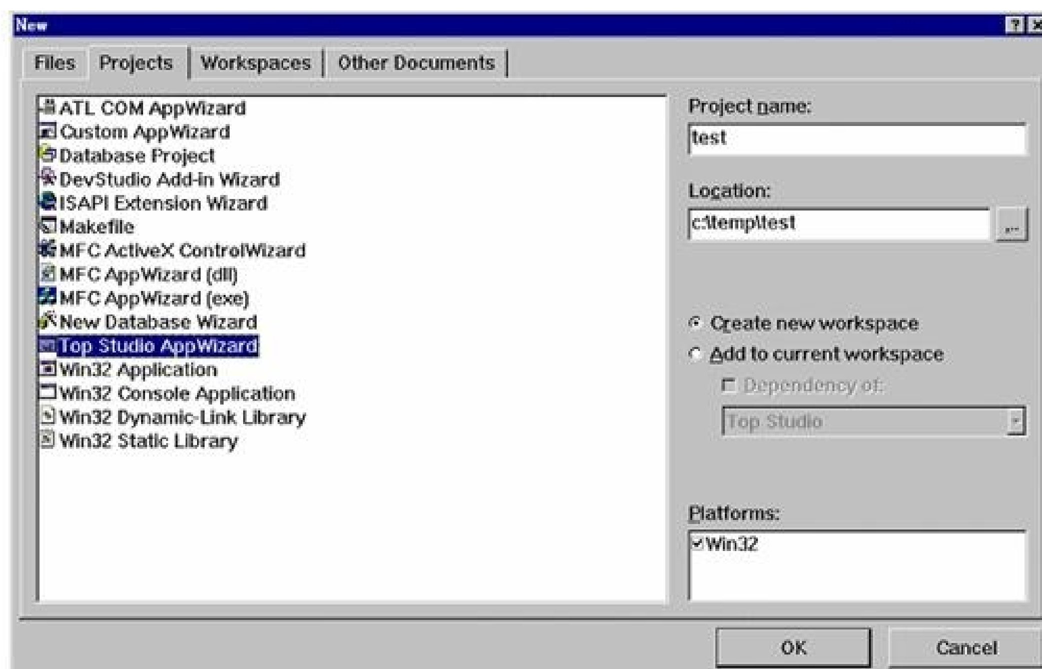
再按下【OK】钮，开始产生程序代码。然后点选整合环境中的【Build/Top Studio.awx】。整合环境下方出现 "Making help file..." 字样。这时候你要注意了，上个厕所喝杯咖啡后它还是那样，一点动静都没有。原来，整合环境启动了 Microsoft Help Workshop，而且把它极小化；你得把它叫出来，让它动作才行。

如果你不想要那些占据很大磁盘空间的 HLP 文件和 HTM 档，也可以把 Microsoft Help Workshop 关掉，控制权便会回到整合环境来，开始进行编译链接的工作。

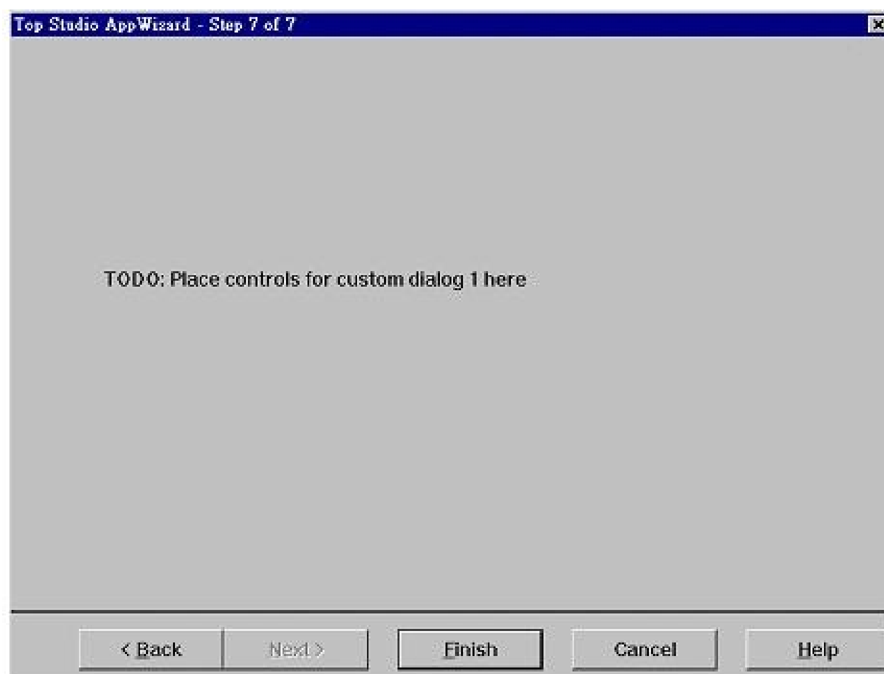
建造过程完毕，我们获得了一个Top Studio.Awx文件。这个文件会被整合环境自动拷贝到 \DevStudio\SharedIDE\Template 磁盘目录中：

```
Directory of E:\DevStudio\SharedIDE\Template
ATL          <DIR>          03-29-97  14:12 ATL
MFC       RCT    4,744      12-04-95  16:09 MFC.RCT
README    TXT    115        10-30-96  17:54 README.TXT
TOPSTU~1  AWX    523,776    04-07-97  17:01 Top Studio.awx
TOPSTU~1  PDB    640,000    04-07-97  17:01 Top Studio.pdb
```

现在，再一次选按整合环境的【File/New】，在【Projects】对话窗中我们看到 Top Studio AppWizard 出现了：



试试它的作用。请像使用一般的MFC AppWizard 那样使用它（像第 4 章那样），你会发现它有 7 个步骤。前 6 个和MFC AppWizard 完全一样，第 7 个画面如下：



哇喔，怎么会这样？当然是这样，因为你还没有做任何程序动作嘛！目前 Top Studio AppWizard 产生出来的程序代码和第 4 章的Scribble step0 完全相同。

剖析 AppWizard Components

图15-2是AppWizard components 的架构图。所谓AppWizard components，就是架构出一个AppWizard 的所有「东西」，包括：

1. Dialog Templates（Dialog Resources）

2. Dialog Classes

3. Text Templates（Template 子目录中的所有 .H 檔和 .CPP 檔）

4. Macro Dictionary

5. Information Files

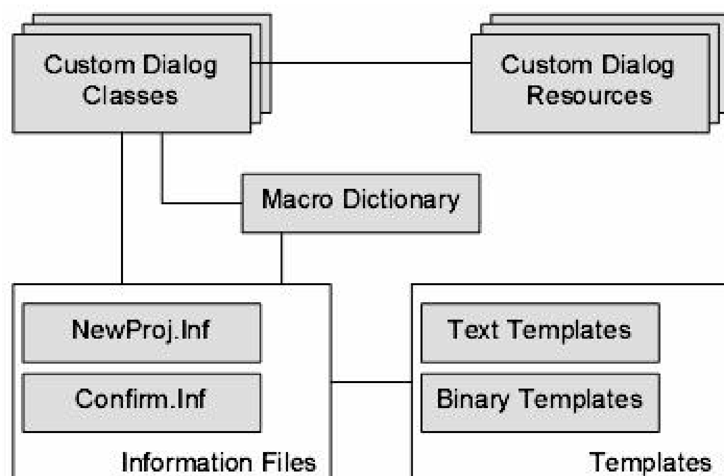


图15-2 用以产生一个custom AppWizard的各种components

Dialog Templates 和 Dialog Classes

以Top Studio AppWizard 为例，由于多出一个对话窗画面，我们势必需要产生一个对话窗面板（template），还要为这面板产生一个对应的C++ 类，并以 DDX/DDV（第10章）取得使用者的输入数据。这些技术我们已经在第10章中学习过。

获得的使用者输入数据如何放置到程序代码产生器所产生的项目原始代码中？

喔，到底谁是程序代码产生器？老实说我也没有办法明确指出是哪个模块，哪个文件（也许就是 AWX 本身）。但是我知道，程序代码产生器会读取 .AWX 文件，做出适当的原始代码来。而 .AWX 不正是前面才刚由Custom AppWizard 做出来吗？里面有些什么蹊跷呢？是的，有许多所谓的macros和directives存在于 Custom AppWizard 所产生的"text template"（也就是template子目录中的所有 .CPP 和 .H 檔）中。以Top Studio AppWizard 为例，我们获得这些文件：

```
H:\U004\PROG\TOP.15：
Top Studio.h
StdAfx.h
Top StudioAw.h
Debug.h
Resource.h
Chooser.h
Cstm1Dlg.h <---- 稍后要修改此文件内容
Top Studio.cpp
StdAfx.cpp
Top StudioAw.cpp
Debug.cpp
Chooser.cpp
Cstm1Dlg.cpp  <---- 稍后要修改此文件内容
H:\U004\PROG\TOP.15\TEMPLATE：<---- 稍后要修改所有这些文件的内容
DlgRoot.h
Dialog.h
Root.h
StdAfx.h
Frame.h
ChildFrm.h
Doc.h
View.h
RecSet.h
SrvrItem.h
IpFrame.h
CntrItem.h
DlgRes.h
Resource.h
DlgRoot.cpp
Dialog.cpp
Root.cpp
StdAfx.cpp
Frame.cpp
ChildFrm.cpp
Doc.cpp
View.cpp
RecSet.cpp
SrvrItem.cpp
IpFrame.cpp
CntrItem.cpp
NewProj.inf
Confirm.inf
```

Macros

我们惯常所说的程序中的 macro，通常带有「动作」。这里的 macro 则是用来代表一个常数。前后以包夹起来的字符串即为一个macro 名称，例如：

```
class $$FRAME_CLASS$$ : public $$FRAME_BASE_CLASS$$
```

程序代码产生器看到这样的句子，如果发现$$FRAME_CLASS$$ 被定义为"CMDIFrameWnd"，$$FRAME_BASE_CLASS$$ 被定义为 "CFrameWnd"，就产生出这样的句子：

```
class CMDIFrameWnd : public CFrameWnd
```

Developer Studio 系统已经内建一组标准的 macros 如下，给 AppWizard 所产生的每一个项目使用：

```
宏名称              意义
APP              应用程序的CWinApp-driven class.
FRAME            应用程序的main frame class.
DOC              应用程序的document class.
VIEW             应用程序的view class.
CHILD_FRAME      应用程序的MDI child frame class (如果有的话)
DLG              应用程序的main dialog box class (在dialog-based 程序中)
RECSET           应用程序的recordset class (如果有的话)
SRVRITEM         应用程序的main server-item class (如果有的话)
CNTRITEM         应用程序的main container-item class (如果有的话)
IPFRAME          应用程序的in-place frame class (如果有的话)
```

另外还有一组macro，可以和前面那组搭配运用：

```
宏名称              意义
class              类名称 (小写)
CLASS              类名称 (大写)
base_class         基类的名称 (小写)
BASE_CLASS         基类的名称 (大写)
ifile              实现文件名称 (.CPP 文件，不含扩展名) (小写)
IFILE              实现文件名称 (.CPP 文件，不含扩展名) (大写)
Hfile              表头文件名称 (.H 文件，不含扩展名) (小写)
hFILE              表头文件名称 (.H 文件，不含扩展名) (大写)
ROOT               应用程序的项目名称 (全部大写)
root               应用程序的项目名称 (全部小写)
Root               应用程序的项目名称 (可以引大小写)
```

图 15-3 列出项目名称为 Scribble 的某些个标准宏内容。

宏	实际内容
APP_CLASS	CScribbleApp
VIEW_IFILE	SCRIBBLEVIEW
DOC_HFILE	SCRIBBLEDOC
doc_hfile	scribbledoc
view_hfile	scribbleview

图15-3 项目名称为Scribble的数个标准宏内容

Directives

所谓directives，类似程序语言中的条件控制句（像是if、else 等等），用来控制text templates 中的流程。字符串前面如果以$$开头，就是一个 directive，例如：

```
$$IF(PROJTYPE_MDI)
...
$$ELSE
...
$$ENDIF
```

每一个 directive 必须出现在每一行的第一个字符。

系统提供了一组标准的 directives 如下：

```
$$IF
$$ELIF
$$ELSE
$$ENDIF
$$BEGINLOOP
$$ENDLOOP
$$SET_DEFAULT_LANG
$$//
$$INCLUDE
```

动手修改 Top Studio AppWizard

我的目的是做出一个属于我个人研究室专用的 Top Studio AppWizard，以原本的MFC AppWizard（exe）为基础，加上第 7 个步骤，让程序员填入姓名、简易说明，然后Top Studio AppWizard 就能够把这些数据加到每一个原始代码文件最前端。

看来我们已经找到出口了。我们应该先为Top Studio AppWizard 产生一个对话窗，当做步骤 7 的画面，再产生一个对应的C++ 类，于是DDX功能便能够取得对话窗所接收到的输入字符串（程序员姓名和程序主旨）。然后我们设计一些macros，再撰写一小段代码（其中用到那些macros），把这一小段代码加到每一个 .CPP 和 .H 檔的最前面。大功告成。

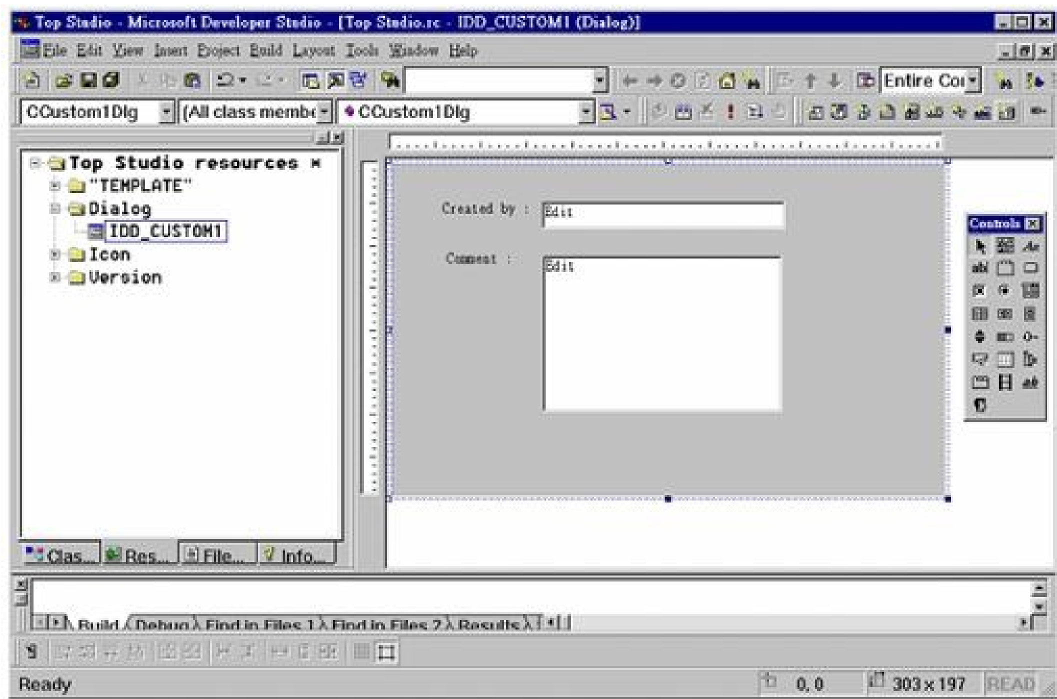
本例不需要我们动手写directives。

我想我遗漏了一个重要的东西。Macros 如何定义？放在什么地方？我曾经在本书第 8 章介绍 Scribble 的数据结构时，谈到collection classes。其中有一种数据结构名为 Map（也就是 Dictionary）。Macros 正是被定义并储存在一个Map之中，并以macro 名称做为键值（key）。

让我们一步一步来。

利用资源编辑器修改IDD_CUSTOM1对话窗画面

请参考第 4 章和第10章，修改IDD_CUSTOM1 对话窗画面如下：



两个edit控制组件的ID如图15-4所示。

利用ClassWizard修改IDD_CUSTOM1对话窗的对应类CCustom1Dlg

图15-4列出每一个控制组件的类型、识别代码及其对应的变量名称等数据。变量将做为DDX所用。修改动作如图15-5。

```
control ID          名称          种类       变量类型
IDC_EDIT_AUTHOR    m_szAuthor    Value      CString
IDC_EDIT_COMMENT   m_szComment   Value      CString
```

图15-4 IDD_CUSTOM1对话窗控制组件的类型、ID、对应的变量名称

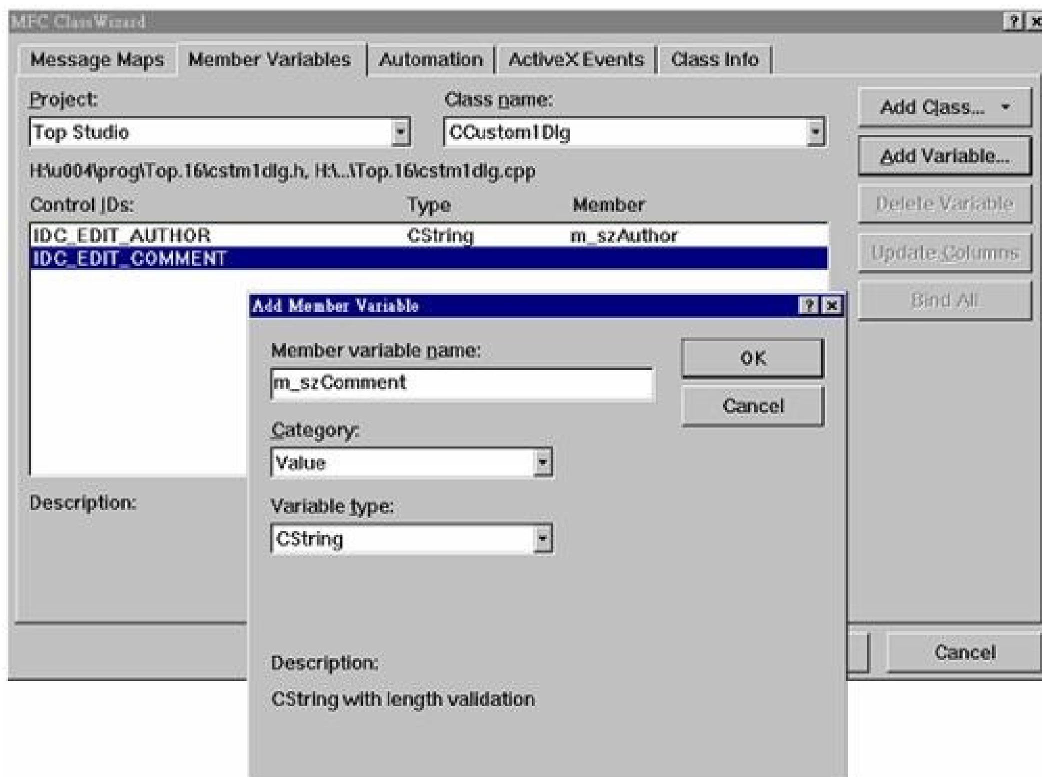


图15-5 利用ClassWizard 为IDD_CUSTOM1对话窗的两个edit

控制组件加上两个对应的变量m_szAuthor和m_szComment，以为DDX 所用

Custom AppWizard 为我们做出来的这个CCustom1Dlg必定派生自 CAppWizStepDlg。你不会在 MFC 类架构文件中发现 CAppWizStepDlg，它是 Visual C++ 的 mfcapwz.dll所提供的一个类。此类有一个虚函数 OnDismiss，当AppWizard 的使用者选按【Back】或【Next】或【Finish】钮时就会被唤起。如果它传回TRUE，AppWizard 就可以切换对话窗；如果传回的是FALSE，就不能。我们可以在这个函数中做数值检验的工作，更重要的是做 macros的设定工作。

改写OnDismiss虚函数，在其中定义macros

前面我已经说过，macros 的定义储存在一个Map 结构中。它在哪里？

整个Top Studio AppWizard（以及其它所有的 custom AppWizard）的主类系派生自系统提供的CCustomAppWiz：

```
// in Top StudioAw.h
class CTopStudioAppWiz : public CCustomAppWiz
{
    ....
};
// in "Top StudioAw.cpp"
CTopStudioAppWiz TopStudioaw;  // 类似 application object。
// 对象命名规则是 "项目名称" + "aw"。
```

你不会在MFC类架构文件中发现CCustomAppWiz，它是Visual C++的 mfcapwz.dll所提供的一个类。此类拥有一个CMapStringToString 对象，名为m_Dictionary，所以TopStudioaw 自然就继承了m_Dictionary。这便是储存macros 定义的地方。我们可以利用 TopStudioaw.m_Dictionary[xxx] = xxx 的方式来加入一个个的macros。

现在，改写OnDismiss 虚函数如下：

```
#0001 // This is called whenever the user presses Next, Back, or Finish with this step
#0002 //  present.  Do all validation & data exchange from the dialog in this function.
#0003 BOOL CCustom1Dlg::OnDismiss()
#0004 {
#0005     if (!UpdateData(TRUE))
#0006         return FALSE;
#0007
#0008     if( m_szAuthor.IsEmpty() == FALSE )
#0009         TopStudioaw.m_Dictionary["PROJ_AUTHOR"] = m_szAuthor;
#0010     else
#0011         TopStudioaw.m_Dictionary["PROJ_AUTHOR"] = "";
#0012
#0013     if( m_szComment.IsEmpty() == FALSE )
#0014         TopStudioaw.m_Dictionary["PROJ_COMMENT"] = m_szComment;
#0015     else
#0016         TopStudioaw.m_Dictionary["PROJ_COMMENT"] = "";
#0017
#0018     CTime date = CTime::GetCurrentTime();
#0019     CString szDate = date.Format( "%A, %B %d, %Y" );
#0020     TopStudioaw.m_Dictionary["PROJ_DATE"] = szDate;
#0021
#0022     return TRUE;  // return FALSE if the dialog shouldn't be dismissed
#0023 }
```

这么一来我们就定义了三个 macros：

```
macro名称                macro内容
PROJ_AUTHOR               m_szAuthor
PROJ_DATE                 szDate
PROJ_COMMENT              m_szComment
```

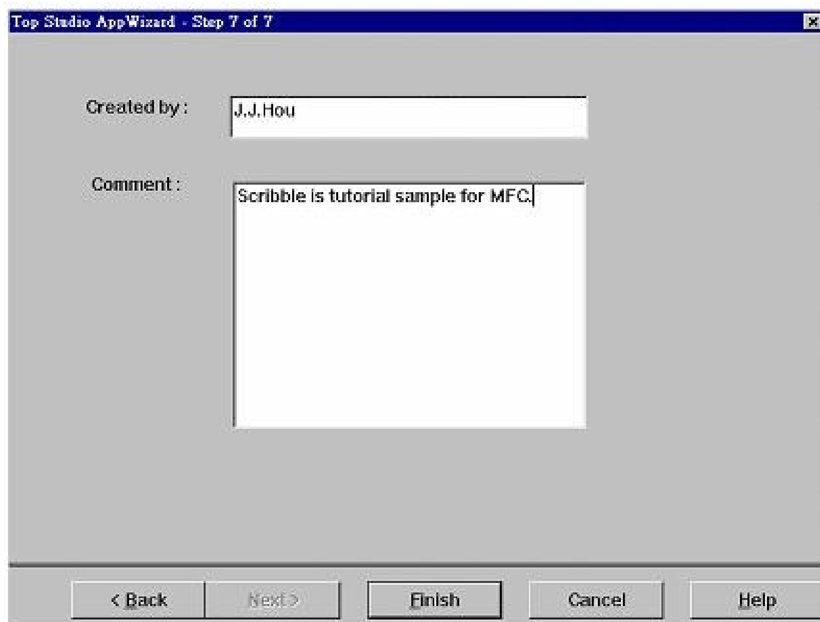
修改text template

现在，为Top Studio AppWizard 的template 子目录中的每一个 .H 档和 .CPP 档增加一小段代码，放在文件最前端：

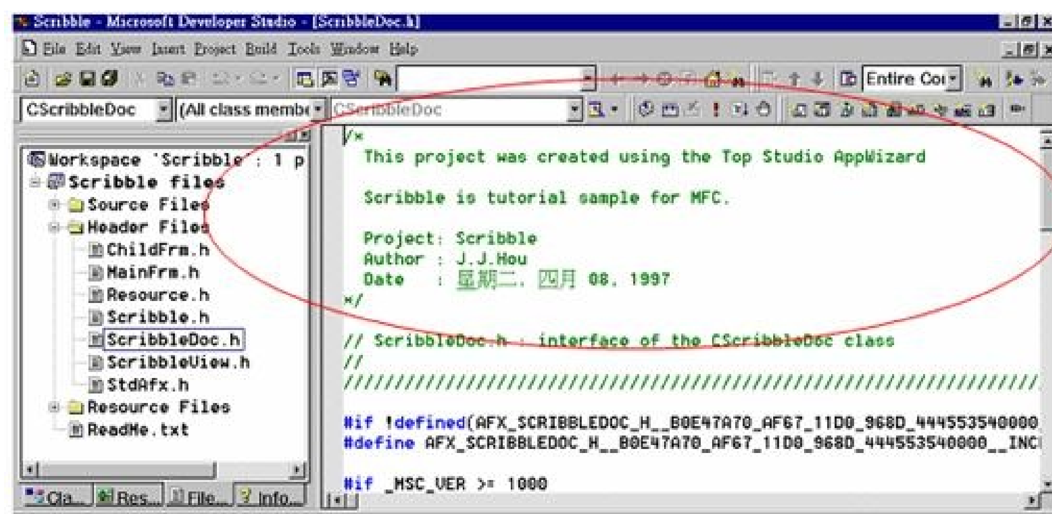
```
/*
This project was created using the Top Studio AppWizard
$$PROJ_COMMENT$$
Project: $$Root$$
Author : $$PROJ_AUTHOR$$
Date : $$PROJ_DATE$$
*/
```

Top Studio AppWizard 执行结果

重新编译链接，然后使用 Top Studio AppWizard 产生一个项目。第 7 个步骤的画面如下：



由Top Studio AppWizard 产生出来的程序代码中，每一个 .CPP 和 .H 檔最前面果然有下面数行文字，大功告成。



更多的信息

我在本章中只是简单示范了一下「继承自原有之 Wizard，再添加新功能」的作法。这该算是半自助吧。全自助的作法就复杂许多。Walter Oney 有一篇 "Pay No Attention to the Man Behind the Curtain! Write Your Own C++ AppWizards" 文章，发表于Microsoft Systems Journal的1997 三月号，里面详细描述了全自助的作法。请注意，他是以Visual C++ 4.2 为演练对象。不过，除了画面不同，技术上完全适用于 Visual C++ 5.0。

Dino Esposito 有一篇文章"a new assistant"，发表于Windows Tech Journal的1997 三月号，也值得参考。1997 年五月份的Dr. Dobb's Journal也有一篇名为"Extending Visual C++ : Custom AppWizards make it possible" 的文章，作者是John Roberts。

第16章 站上众人的肩膀 -- 使用Components&ActiveX Controls

从Visual Basic开始，可以说一个以components（软件组件）为中心的程序设计时代，逐渐拉开了序幕。随后Delphi和C++ Builder 陆续登场。Visual Basic使用VBX（Visual Basic eXtension）组件，Delphi和C++ Builder 使用VCL（Visual Component Library）组件，Visual C++ 则使用OCX（OLE Control eXtension）组件。如今 OCX 又演化到所谓ActiveX 组件（其实和OCX 大同小异）。

Microsoft 的Visual Basic（使用Basic 语言），Borland 的Delphi（使用Pascal 语言），以及Borland 的 C++ Builder（使用 C++ 语言），都称得上是一种快速开发工具（RAD，Rapid Application Development）。它们所使用的组件都是 PME（Properties-Method-Event）架构。这使得它们的整合环境（IDE）能够做出非常可视化的开发工具，以拖放、填单的方式完成绝大部份的程序设计工作。它们的应用程序开发程序大约是这个样子：

1. 选择一些适当的软件组件（VBX或VCL）。

2. 打开一个form，把那些软件组件拖放到form 中适当的位置。

3. 在Properties 清单中填写适当的属性。例如精确位置、宽度高度、或是让 A 组件的某个属性连接到 B 组件...等等。

4. 撰写程序代码（method），做为某种event发生时的处理例程。

依我的看法，Visual C++ 还不能够算是RAD。虽然，MFC程序所能够使用的 OCX 也是PME（Properties-Method-Event）架构，但 Visual C++ 整合环境没有能够提供适当工具让我们以那么可视化的方式（像VB或Delphi或C++ Builder 那样拖放、填单）就几乎完成一个程序。

什么是 Component Gallery

Component Gallery 是自从Visual C++ 4.0 之后，整合环境中新增的一个东西。你可以把它想象成一个数据库，储存着ActiveX controls 和可重复使用的 C++ 类（也就是本章所谓的components）。VC++ 5.0的Component Gallery 的使用接口和 VC++ 4.x 有某种程度的不同，不过操控原则基本上是一致的。

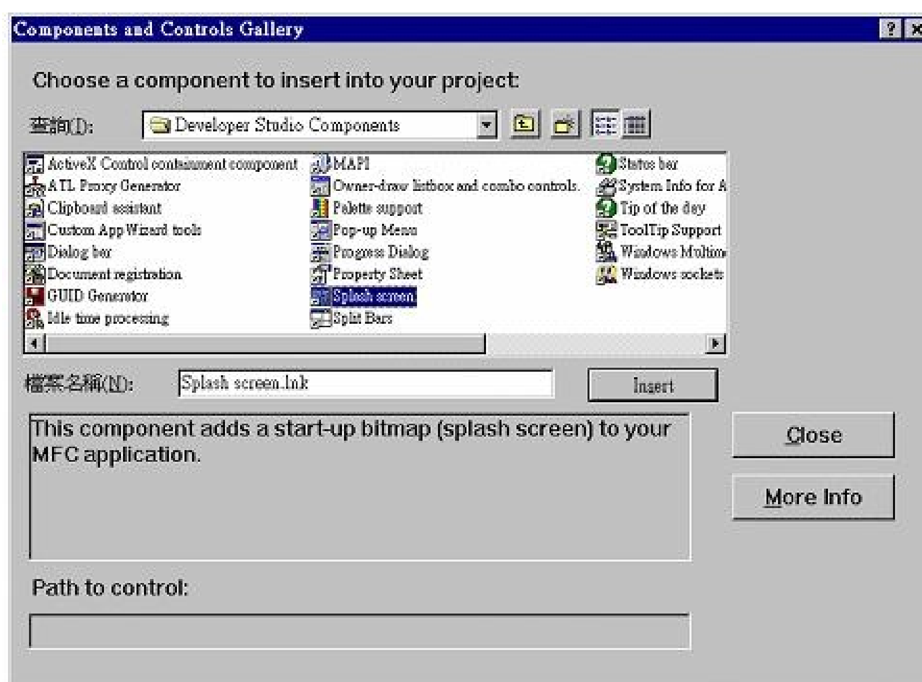
当你安装了Visual C++ 5.0，Component Gallery 已经内含了一些微软所提供的components 和ActiveX controls（注：以下我将把这两样东西统称为「组件」）。选按整合环境的【Project / Add To Project / Components and Controls...】选单项目，你就可以看到Component Gallery：



其中有Developer Studio Components 和 Registered ActiveX Controls 两个数据夹，打开任何一个，就会出现目前系统所拥有的「货色」：

如果你以为这些组件储存在两个地方（一个是它本来的位置，另一份拷贝放在 Component Gallery 之中），那你就错了。Component Gallery 只是存放那些组件的位置数据而已。你可以说，只是存放一个「链接」而已。

为什么组件在此分为Components和ActiveX controls 两种？有什么不同。简单地说，Components 是一些已写好的C++ 类。基本上C++ 类本来就具有重复使用性，Component Gallery 只是把它们多做一些必要的包装，连同其它资源放在一起成为一个包裹。当你需要某个component，Component Gallery 给你的是该components 的原始代码。



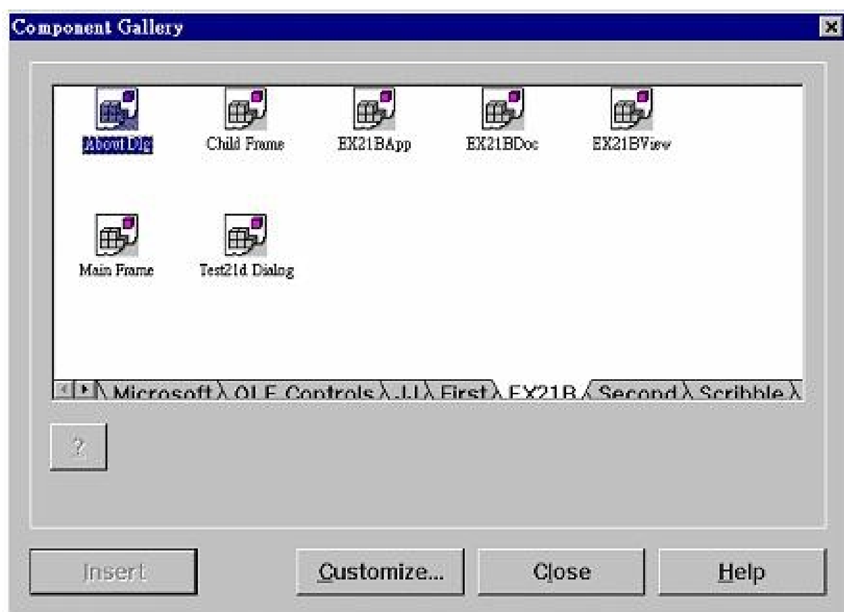
ActiveX controls 不一样。当你选用某个ActiveX controls，Component Gallery 当然也会为你填入一些代码，但它们不是组件的本体。那些代码只是使用组件时所必须的代码，组件本身在 .OCX 文件中（通常注册后的OCX 文件都放在 Windows\System 磁盘子目录）。

ActiveX controls 是很完整的一个有着PME（Proterties-Method-Event）架构的控制组件，但一般欲被重复使用的C++ 类却不会有那么完整的设计或包装。要把一个C++ 类做成完好的包装，放到Component Gallery 中，它必须变为一个单一文件，内含类资讯以及任何必须的资源。这在过去的Visual C++ 4.x中是很容易的事情，因为每次你使用ClassWizard 新增一个类，就有一个核示盒询问你要不要加到Component Gallery：



Visual C++ 4.x 的ClassWizard 新增类对话窗

但这一选项已在Visual C++ 5.0 中拿掉（你可以在第 10 章增加对话窗类时看到新的画面）。看来似乎要增加components 不再是那么方便了。这倒也不是坏事，我想许多人在设计程序时忽略了上图那个选项，于是每一个项目中的每一个类，都被包装到Component Gallery 去，而其中许多根本是没有价值的：



Visual C++ 4.x 的Component Gallery。常常因为程序员的疏忽，而产生了一大堆没有价值的components。

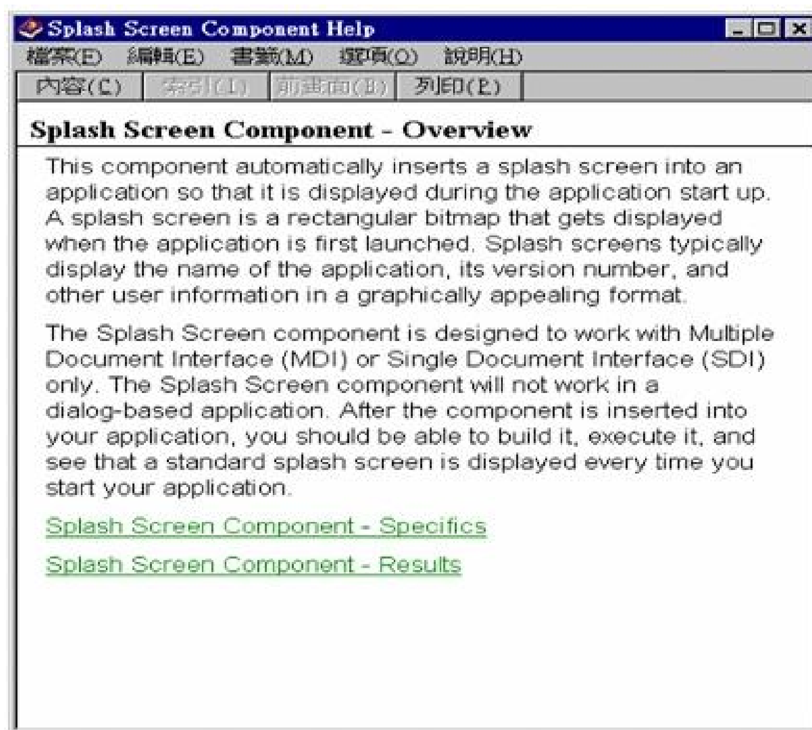
使用 Components

当你选择Component Gallery 中的Developer Studio Components 数据夹，出现许多的components。面对形形色色的「货」，你的心里一定嘀咕着：怎么用嘛？幸好画面上有一个【More Info】按钮，可以提供你比较多的信息。以下我挑三个最简单的components做示范。

Splash screen

所谓Splash Screen，你可以说它是一个「炫耀画面」。玩过微软的Office 吗？每一个Office软件一出场，在它做初始化的那段时间里，都会出现一个画面，就是Splash screen。

Splash Screen 的【More Info】出现这样的画面：



选按上图下方的"Splash Screen Component - Specifics"，你会获得一张使用规格说明，大意如下：

欲插入 splash Screen component，你必须：

1. 打开你希望安插Splash Screen component的那个项目。

2. 选择整合环境中的【Project/Add To Project/Components and Controls】选单项目。

3. 选择"Developer's Studio Components" 数据夹。

4. 选择数据夹中的 Splash Screen component 并按下【Insert】钮。

5. 设定必要的 Splash Screen 选项然后按下【OK】钮。

6. 重建（重新编译链接）项目。

如果要把 Splash Screen 加到一个以对话窗为主（dialog-based）的程序中，你必须在插入这个component之后做以下事情：

1. 找到你的 InitInstance 函数。

2. 在你调用：

```
int nResponse = dlg.DoModal();
```

之前，加上一行：

```
spl.ShowSplashScreen(FALSE);
```

增加这一行代码，可以确保 Splash Screen 在主对话窗被显示之前，会被清除掉。

看来很简单的样子

System Info for About Dlg

看过 WordPad 的【About】对话窗吗：



如果你也想让自己的对话窗有点系统信息的显示能力，可以采用 Component Gallery 提供的这个System Info for About Dlg component。它的规格说明文字如下：

SysInfo component可以为你的程序的About对话窗中加上一些系统信息（可用内存数量以及磁盘剩余空间）。你的程序必须以MFC AppWizard 完成。请参考 WordPad说明文件以获得更多信息。这份规格书不够详细。稍后我会在修改程序代码时加上我自己的说明。

Tip of the Day

看过这种画面吗（微软的Office软件就有）：



这就是「每日小秘诀」。Component Gallery 提供的Tips for the Day component 让你很方便地为自己加上「每日小秘诀」。这个component的使用规格是：

小秘诀文字文件（TIPS.TXT）：

拥有Tips for the Day component的程序将搜寻磁盘中的工作子目录，企图寻找TIPS.TXT 读取秘诀内容。如果你希望这个秘诀文字文件有不同的名称或是放在不同的位置，你可以修改CTIP.CPP中的CTIP 类构造函数。CTIP是预设的类名称。

（侯俊杰注：最后这句话是错误的。我使用这个 component，接受所有的预设项目，获得的类名称却是 CTIPDLG，文件则为 TIPDLG.CPP）

- TIPS.TXT 的格式如下：

  1. 文件必须是ASCII 文字，每一个秘诀以一行文字表示。

  2. 如果某一行文字以分号（;）开头，表示这是一行说明文字，不生实效。说明文字必须有自己单独的一行。

  3. 空白行会被忽略。

  4. 每一个小秘诀最多 1000 个字符。

  5. 每一行不能够以空白或定位符号（tab）开始。

- 小秘诀显示次序：

  预设情况下，小秘诀的出现次序和它们在文件中的排列次序相同。如果全部都出现过了，就再循环一遍。如果文件被更改过了，显示次序就会从头开始。

- 错误情况：

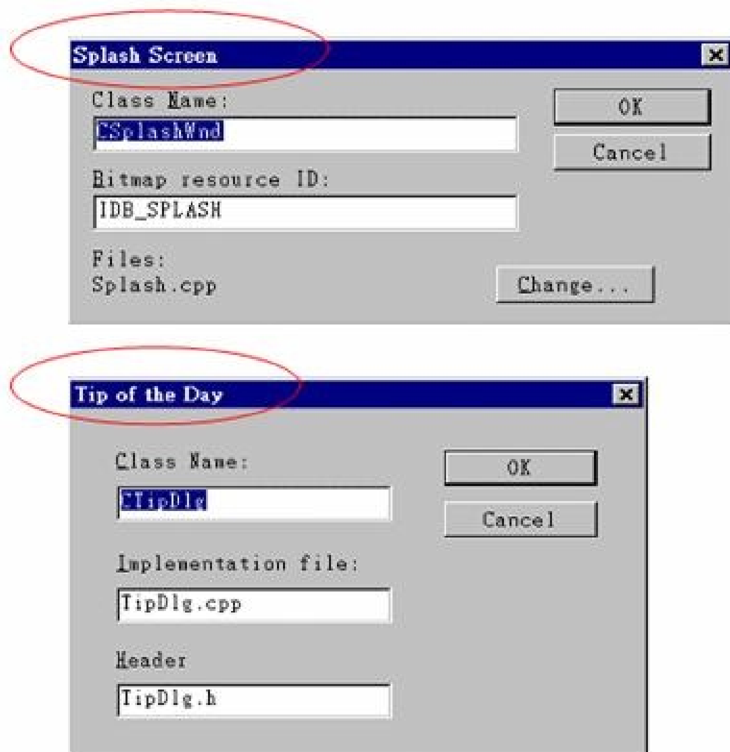
  这个组件希望在MFC程序中被使用。你的程序应该只有一个派生自 CWinApp的类。如果有许多个CWinApp 派生类，此组件会选择其中第一个做为实现的对象。其他的错误情况包括秘诀文字文件不存在，或格式不对等等。

- 在程序的【Help】选单中加上 Tip of The Day 项目：

  这个组件会修改主框窗口的 OnInitMenu 函数，并且在你的【Help】选单下加挂一个Tip of The Day 项目。如果你的程序原本没有【Help】选单，此组件就自动为你产生一个。

Components 实际运用：ComTest 程序

现在，动手吧。首先利用MFC AppWizard 产生一个项目，就像第 4 章的 Scribble step0 那样。我把它命名为ComTest（放在书附光盘的ComTest.17 子目录中）。然后，不要离开这个项目，启动Component Gallery，进入 Developer Studio Components数据夹，分别选择 Splash Screen 和System Info for About Dlg和Tips of the Day三个组件，分别按下【Insert】钮。Splash Screen 和 Tips of the Day 组件会要求我们再指定一些消息：



新增文件

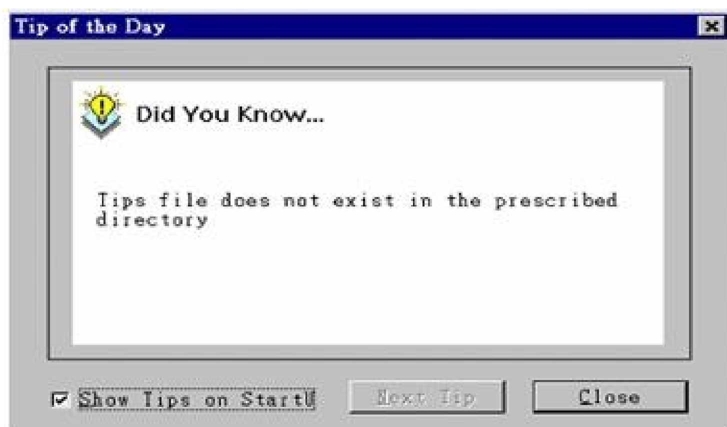
这时候 ComTest 项目中的原始代码有了一些变动（被 Component Gallery 改变）。被改变的文件是：

```
STDAFX.H
RESOURCE H
COMTEST.H
COMTEST.CPP
COMTEST.RC
MAINFRM.H
MAINFRM.CPP
SPLASH.H
SPLASH.CPP
SPLSH16.BMP
TIPDLG.CPP
TIPDLG.H
```

选按整合环境的【Build/ Build ComTest.Exe】，把这个程序建造出来。建造完毕试执行之，你会发现在主窗口出现之前，一开始先有一张画面显现：



然后是每日小秘诀：



然后才是主窗口。至于About对话窗，画面如下（没啥变化）：



看来，我们只要修改一下Splash Screen 画面，并增加一个TIPS.TXT 文字文件，再变化一下About 对话窗，就成了。程序编修动作的确很简单，不过我还是要把这三个组件加诸于你的程序的每一条痕印都揭发出来。

相关变化

让我们分析分析Component Gallery为我们做了些什么事情。

STDAFX.H（阴影部份为新增内容）

```
...
#include <afxwin.h> // MFC core and st
#include <afxext.h> // MFC extensions
#include <afxdisp.h> // MFC OLE automat
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h> // MFC
#endif // _AFX_NO_AFXCMN_SUPPORT
#include <H:\u002p\prog\ComTest.16\TipDlg.h>
...
```

RESOURCE.H

下面是针对三个组件新增的一些常数定义。凡是稍后修改程序时会用到的常数，我都加上批注，提醒您特别注意。

```
...
#define IDB_SPLASH 102 //Splash screen 所加，代表一张16色bitmap 画面
#define CG_IDS_PHYSICAL_MEM 103
#define CG_IDS_DISK_SPACE 104
#define CG_IDS_DISK_SPACE_UNAVAIL 105
#define IDB_LIGHTBULB 106
#define IDD_TIP 107
#define CG_IDS_TIPOFTHEDAY 108//Tips 所加，一个字符串。稍后我要把它改为中文内容。
#define CG_IDS_TIPOFTHEDAYMENU 109
#define CG_IDS_DIDYOUKNOW 110//Tips 所加，一个字符串。稍后我要把它改为中文内容
#define CG_IDS_FILE_ABSENT 111
#define CG_IDP_FILE_CORRUPT 112
#define CG_IDS_TIPOFTHEDAYHELP 113
#define IDC_PHYSICAL_MEM 1000 //SysInfo 所加，代表「可用内存」这个static字段
#define IDC_BULB 1000
#define IDC_DISK_SPACE 1001 // SysInfo所加，代表「磁盘剩余空间」这个static字段
#define IDC_STARTUP 1001
#define IDC_NEXTTIP 1002
#define IDC_TIPSTRING 1004
...
COMTEST.H（阴影部份为新增内容）
class CComTestApp : public CWinApp
{
public:
    virtual BOOL PreTranslateMessage(MSG* pMsg);
    CComTestApp();
    ...
private:
    void ShowTipAtStartup(void);
private:
    void ShowTipOfTheDay(void);
}
```

COMTEST.CPP（阴影部份为新增内容）

```
#0001  ...
#0002  #include "Splash.h"
#0003  #include <dos.h>
#0004  #include <direct.h>
#0005
```

```
#0006 BEGIN_MESSAGE_MAP(CComTestApp, CWinApp)
#0007     ON_COMMAND(CG_IDS_TIPOFTHEDAY, ShowTipOfTheDay)
#0008     //{{AFX_MSG_MAP(CComTestApp)
#0009     ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
#0010         // NOTE - the ClassWizard will add and remove mapping macros here.
#0011         //    DO NOT EDIT what you see in these blocks of generated code!
#0012     //}}AFX_MSG_MAP
#0013     // Standard file based document commands
#0014     ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
#0015     ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
#0016     // Standard print setup command
#0017     ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
#0018 END_MESSAGE_MAP()
#0019
#0020 BOOL CComTestApp::InitInstance()
#0021 {
#0022     // CG: The following block was added by the Splash Screen component.

#0023     {
#0024       CCommandLineInfo cmdInfo;
#0025       ParseCommandLine(cmdInfo);
#0026       CSplashWnd::EnableSplashScreen(cmdInfo.m_bShowSplash);
#0027     }
#0028
#0029     AfxEnableControlContainer();
#0030     ...
#0031
#0032     // CG: This line inserted by 'Tip of the Day' component.
#0033     ShowTipAtStartup();
#0034
#0035     return TRUE;
#0036 }
#0037 ...
#0038 BOOL CComTestApp::PreTranslateMessage(MSG* pMsg)
#0039 {
#0040     // CG: The following lines were added by the Splash Screen component.
#0041     if (CSplashWnd::PreTranslateAppMessage(pMsg))
#0042         return TRUE;
#0043
#0044     return CWinApp::PreTranslateMessage(pMsg);
#0045 }
#0046
#0047 BOOL CAboutDlg::OnInitDialog()
#0048 {
#0049     CDialog::OnInitDialog();  // CG: This was added by System Info Component.
#0050
#0051     // CG: Following block was added by System Info Component.

#0052     {
#0053         CString strFreeDiskSpace;
#0054         CString strFreeMemory;
#0055         CString strFmt;
#0056
#0057         // Fill available memory
#0058         MEMORYSTATUS MemStat;
#0059         MemStat.dwLength = sizeof(MEMORYSTATUS);
#0060         GlobalMemoryStatus(&MemStat);
#0061         strFmt.LoadString(CG_IDS_PHYSICAL_MEM);
#0062         strFreeMemory.Format(strFmt, MemStat.dwTotalPhys / 1024L);
#0063
#0064         //TODO: Add a static control to your About Box to receive the memory
#0065         //      information.  Initialize the control with code like this:
#0066         // SetDlgItemText(IDC_PHYSICAL_MEM, strFreeMemory);
#0067
#0068         // Fill disk free information
```

```
#0069        struct _diskfree_t diskfree;
#0070        int nDrive = _getdrive(); // use current default drive
#0071        if (_getdiskfree(nDrive, &diskfree) == 0)
#0072        {
#0073            strFmt.LoadString(CG_IDS_DISK_SPACE);
#0074            strFreeDiskSpace.Format(strFmt,
#0075                    (DWORD)diskfree.avail_clusters *
#0076                    (DWORD)diskfree.sectors_per_cluster *
#0077                    (DWORD)diskfree.bytes_per_sector / (DWORD)1024L,
#0078                    nDrive-1 + _T('A'));
#0079        }
#0080        else
#0081            strFreeDiskSpace.LoadString(CG_IDS_DISK_SPACE_UNAVAIL);
#0082
#0083        //TODO: Add a static control to your About Box to receive the memory
#0084        //      information.  Initialize the control with code like this:
#0085        // SetDlgItemText(IDC_DISK_SPACE, strFreeDiskSpace);
#0086    }
#0087
#0088    return TRUE;    // CG:  This was added by System Info Component.
#0089 }
```

COMTEST.RC（阴影部份为新增内容）

```
IDB_SPLASH  BITMAP  DISCARDABLE  "Splsh16.bmp"
...
IDD_TIP DIALOG DISCARDABLE  0, 0, 231, 164
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Tip of the Day"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL         "",-1,"Static",SS_BLACKFRAME,12,11,207,123
    LTEXT           "Some String",IDC_TIPSTRING,28,63,177,60
    CONTROL         "&Show Tips on StartUp",IDC_STARTUP,"Button",
                    BS_AUTOCHECKBOX | WS_GROUP | WS_TABSTOP,13,146,85,10
    PUSHBUTTON      "&Next Tip",IDC_NEXTTIP,109,143,50,14,WS_GROUP
    DEFPUSHBUTTON   "&Close",IDOK,168,143,50,14,WS_GROUP
    CONTROL         "",IDC_BULB,"Static",SS_BITMAP,20,17,190,111
END
...
STRINGTABLE DISCARDABLE
BEGIN
    CG_IDS_PHYSICAL_MEM          "%lu KB"
    CG_IDS_DISK_SPACE            "%lu KB Free on %c:"
    CG_IDS_DISK_SPACE_UNAVAIL    "Unavailable"
    CG_IDS_TIPOFTHEDAY           "Displays a Tip of the Day."

    CG_IDS_TIPOFTHEDAYMENU      "Ti&p of the Day..."
    CG_IDS_DIDYOUKNOW           "Did You Know..."
CG_IDS_FILE_ABSENT   "Tips file does not exist in the prescribed directory"
END

STRINGTABLE DISCARDABLE
BEGIN
    CG_IDP_FILE_CORRUPT     "Trouble reading the tips file"
    CG_IDS_TIPOFTHEDAYHELP  "&Help"
END
```

MAINFRM.H（阴影部份为新增内容）

```
class CMainFrame : public CMDIFrameWnd
{
...
// Overrides
       // ClassWizard generated virtual function overrides
       //{{AFX_VIRTUAL(CMainFrame)
       virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
       //}}AFX_VIRTUAL
...
// Generated message map functions
protected:
       afx_msg void OnInitMenu(CMenu* pMenu);
       ...
};
```

MAINFRM.CPP（阴影部份为新增内容）

```
#0001  ...
#0002  #include "Splash.h"
#0003  ...
#0004  int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
#0005  {
#0006      ...
#0007      // CG: The following line was added by the Splash Screen component.
#0008      CSplashWnd::ShowSplashScreen(this);
#0009
#0010      return 0;
#0011  }
#0012  ...
#0013  ///////////////////////////////////////////////////////////////////////
#0014  // CMainFrame message handlers
#0015
```

```
#0016 void CMainFrame::OnInitMenu(CMenu* pMenu)
#0017 {
#0018     CMDIFrameWnd::OnInitMenu(pMenu);
#0019
#0020     // CG: This block added by 'Tip of the Day' component.
#0021     {
#0022         // TODO: This code adds the "Tip of the Day" menu item
#0023         // on the fly.  It may be removed after adding the menu
#0024         // item to all applicable menu items using the resource
#0025         // editor.
#0026
#0027         // Add Tip of the Day menu item on the fly!
#0028         static CMenu* pSubMenu = NULL;
#0029
#0030         CString strHelp; strHelp.LoadString(CG_IDS_TIPOFTHEDAYHELP);
#0031         CString strMenu;
#0032         int nMenuCount = pMenu->GetMenuItemCount();
#0033         BOOL bFound = FALSE;
#0034         for (int i=0; i < nMenuCount; i++)
#0035         {
#0036             pMenu->GetMenuString(i, strMenu, MF_BYPOSITION);
#0037             if (strMenu == strHelp)
#0038             {
#0039                 pSubMenu = pMenu->GetSubMenu(i);
#0040                 bFound = TRUE;
```

```
#0041                 ASSERT(pSubMenu != NULL);
#0042             }
#0043         }
#0044
#0045         CString strTipMenu;
#0046         strTipMenu.LoadString(CG_IDS_TIPOFTHEDAYMENU);
#0047         if (!bFound)
#0048         {
#0049             // Help menu is not available. Please add it!
#0050             if (pSubMenu == NULL)
```

```
#0051             {
#0052                 // The same pop-up menu is shared between mainfrm and
frame
#0053                 // with the doc.
#0054                 static CMenu popUpMenu;
#0055                 pSubMenu = &popUpMenu;
#0056                 pSubMenu->CreatePopupMenu();
#0057                 pSubMenu->InsertMenu(0, MF_STRING|MF_BYPOSITION,
#0058                         CG_IDS_TIPOFTHEDAY, strTipMenu);
#0059             }
#0060             pMenu->AppendMenu(MF_STRING|MF_BYPOSITION|MF_ENABLED|MF_POPUP,
```

```
#0061                    (UINT)pSubMenu->m_hMenu, strHelp);
#0062             DrawMenuBar();
#0063         }
#0064         else
#0065         {
#0066             // Check to see if the Tip of the Day menu has already been
added.
#0067             pSubMenu->GetMenuString(0, strMenu, MF_BYPOSITION);
#0068
#0069             if (strMenu != strTipMenu)
#0070             {
#0071                 // Tip of the Day submenu has not been added to the
#0072                 // first position, so add it.
#0073                 pSubMenu->InsertMenu(0, MF_BYPOSITION);  // Separator
#0074                 pSubMenu->InsertMenu(0, MF_STRING|MF_BYPOSITION,
#0075                         CG_IDS_TIPOFTHEDAY, strTipMenu);
#0076             }
#0077         }
#0078     }
#0080  }
```

SPLASH.H（全新内容）

```
#0001  // CG: This file was added by the Splash Screen component.
#0002
#0003  #ifndef _SPLASH_SCRN_
#0004  #define _SPLASH_SCRN_
#0005
#0006  // Splash.h : header file
#0007
#0008  ///////////////////////////////////////////////////////////////////////////
#0009  //   Splash Screen class
#0010
#0011  class CSplashWnd : public CWnd
#0012  {
#0013  // Construction
#0014  protected:
#0015          CSplashWnd();
```

```
#0016
#0017  // Attributes:
#0018  public:
#0019          CBitmap m_bitmap;
#0020
#0021  // Operations
#0022  public:
#0023          static void EnableSplashScreen(BOOL bEnable = TRUE);
```

```
#0024         static void ShowSplashScreen(CWnd* pParentWnd = NULL);
#0025         static BOOL PreTranslateAppMessage(MSG* pMsg);
#0026
#0027  // Overrides
#0028         // ClassWizard generated virtual function overrides
#0029         //{{AFX_VIRTUAL(CSplashWnd)
#0030         //}}AFX_VIRTUAL
#0031
#0032  // Implementation
#0033  public:
#0034         ~CSplashWnd();
#0035         virtual void PostNcDestroy();
#0036
#0037  protected:
#0038         BOOL Create(CWnd* pParentWnd = NULL);
#0039         void HideSplashScreen();
#0040         static BOOL c_bShowSplashWnd;
#0041         static CSplashWnd* c_pSplashWnd;
#0042
#0043  // Generated message map functions
#0044  protected:
#0045         //{{AFX_MSG(CSplashWnd)
#0046         afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
#0047         afx_msg void OnPaint();
#0048         afx_msg void OnTimer(UINT nIDEvent);
#0049         //}}AFX_MSG
#0050         DECLARE_MESSAGE_MAP()
#0051  };
#0052
#0053  #endif
```

SPLASH.CPP（全新内容）

```
#0001  // CG: This file was added by the Splash Screen component.
#0002  // Splash.cpp : implementation file
#0003
#0004  #include "stdafx.h"  // e. g. stdafx.h
#0005  #include "resource.h"  // e.g. resource.h
#0006
#0007  #include "Splash.h"  // e.g. splash.h
#0008
#0009  #ifdef _DEBUG
#0010  #define new DEBUG_NEW
#0011  #undef THIS_FILE
#0012  static char BASED_CODE THIS_FILE[] = __FILE__;
#0013  #endif

#0014
#0015  /////////////////////////////////////////////////////////////////////////////
```

```
#0016 //   Splash Screen class
#0017
#0018 BOOL CSplashWnd::c_bShowSplashWnd;
#0019 CSplashWnd* CSplashWnd::c_pSplashWnd;
#0020 CSplashWnd::CSplashWnd()
#0021 {
#0022 }
#0023
#0024 CSplashWnd::~CSplashWnd()
#0025 {
#0026        // Clear the static window pointer.
#0027        ASSERT(c_pSplashWnd == this);
#0028        c_pSplashWnd = NULL;
#0029 }
#0030
#0031 BEGIN_MESSAGE_MAP(CSplashWnd, CWnd)
#0032        //{{AFX_MSG_MAP(CSplashWnd)
#0033        ON_WM_CREATE()
#0034        ON_WM_PAINT()
#0035        ON_WM_TIMER()
#0036        //}}AFX_MSG_MAP
#0037 END_MESSAGE_MAP()
#0038
#0039 void CSplashWnd::EnableSplashScreen(BOOL bEnable /*= TRUE*/)
#0040 {
#0041        c_bShowSplashWnd = bEnable;
#0042 }
#0043
#0044 void CSplashWnd::ShowSplashScreen(CWnd* pParentWnd /*= NULL*/)
#0045 {
#0046        if (!c_bShowSplashWnd || c_pSplashWnd != NULL)
```

```
#0047               return;
#0048
#0049        // Allocate a new splash screen, and create the window.
#0050        c_pSplashWnd = new CSplashWnd;
#0051        if (!c_pSplashWnd->Create(pParentWnd))
#0052               delete c_pSplashWnd;
#0053        else
#0054               c_pSplashWnd->UpdateWindow();
#0055 }
#0056
#0057 BOOL CSplashWnd::PreTranslateAppMessage(MSG* pMsg)
#0058 {
#0059        if (c_pSplashWnd == NULL)
```

```
#0060               return FALSE;
#0061
#0062        // If we get a keyboard or mouse message, hide the splash screen.
#0063        if (pMsg->message == WM_KEYDOWN ||
#0064            pMsg->message == WM_SYSKEYDOWN ||
#0065            pMsg->message == WM_LBUTTONDOWN ||
#0066            pMsg->message == WM_RBUTTONDOWN ||
#0067            pMsg->message == WM_MBUTTONDOWN ||
#0068            pMsg->message == WM_NCLBUTTONDOWN ||
#0069            pMsg->message == WM_NCRBUTTONDOWN ||
#0070            pMsg->message == WM_NCMBUTTONDOWN)
```

```
#0071        {
#0072              c_pSplashWnd->HideSplashScreen();
#0073              return TRUE;    // message handled here
#0074        }
#0075
#0076        return FALSE;   // message not handled
#0077  }
#0078
#0079  BOOL CSplashWnd::Create(CWnd* pParentWnd /*= NULL*/)
#0080  {
#0081      if (!m_bitmap.LoadBitmap(IDB_SPLASH))
#0082           return FALSE;
#0083
#0084      BITMAP bm;
#0085      m_bitmap.GetBitmap(&bm);
#0086
#0087      return CreateEx(0,
#0088            AfxRegisterWndClass(0, AfxGetApp()->LoadStandardCursor(IDC_ARROW)),
#0089            NULL, WS_POPUP | WS_VISIBLE, 0, 0, bm.bmWidth, bm.bmHeight,
pParentWnd->GetSafeHwnd(), NULL);
#0090  }
#0091
#0092  void CSplashWnd::HideSplashScreen()
#0093  {
#0094        // Destroy the window, and update the mainframe.
#0095        DestroyWindow();
#0096        AfxGetMainWnd()->UpdateWindow();
#0097  }
#0098
#0099  void CSplashWnd::PostNcDestroy()
#0100  {
#0101        // Free the C++ class.
#0102        delete this;
#0103  }
#0104

#0105  int CSplashWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
#0106  {
#0107        if (CWnd::OnCreate(lpCreateStruct) == -1)
#0108              return -1;
#0109
#0110        // Center the window.
#0111        CenterWindow();
#0112
#0113        // Set a timer to destroy the splash screen.
#0114        SetTimer(1, 750, NULL);
#0115
#0116        return 0;
#0117  }
#0118
#0119  void CSplashWnd::OnPaint()
#0120  {
#0121        CPaintDC dc(this);
#0122
#0123        CDC dcImage;
#0124        if (!dcImage.CreateCompatibleDC(&dc))
#0125              return;
#0126
#0127        BITMAP bm;
#0128        m_bitmap.GetBitmap(&bm);
#0129
#0130        // Paint the image.
#0131        CBitmap* pOldBitmap = dcImage.SelectObject(&m_bitmap);
#0132        dc.BitBlt(0, 0, bm.bmWidth, bm.bmHeight, &dcImage, 0, 0, SRCCOPY);
#0133        dcImage.SelectObject(pOldBitmap);
#0134  }
#0135
#0136  void CSplashWnd::OnTimer(UINT nIDEvent)
#0137  {
#0138        // Destroy the splash screen window.
#0139        HideSplashScreen();
#0140  }
```

TIPDLG.H（全新内容）

```
#0001 #if !defined(TIPDLG_H_INCLUDED_)
#0002 #define TIPDLG_H_INCLUDED_
#0003
#0004 // CG: This file added by 'Tip of the Day' component.
#0005
#0006 /////////////////////////////////////////////////////////////////////////
#0007 // CTipDlg dialog

#0008
#0009 class CTipDlg : public CDialog
#0010 {
#0011 // Construction
#0012 public:
#0013        CTipDlg(CWnd* pParent = NULL);   // standard constructor
#0014
#0015 // Dialog Data
#0016        //{{AFX_DATA(CTipDlg)
#0017        // enum { IDD = IDD_TIP };
#0018        BOOL    m_bStartup;
#0019        CString m_strTip;
#0020        //}}AFX_DATA
#0021
#0022        FILE* m_pStream;
#0023
#0024 // Overrides
#0025        // ClassWizard generated virtual function overrides
#0026        //{{AFX_VIRTUAL(CTipDlg)
#0027        protected:
#0028        virtual void DoDataExchange(CDataExchange* pDX);   // DDX/DDV support
#0029        //}}AFX_VIRTUAL
#0030
#0031 // Implementation
#0032 public:
#0033        virtual ~CTipDlg();
#0034
#0035 protected:
#0036     // Generated message map functions
#0037     //{{AFX_MSG(CTipDlg)
#0038     afx_msg void OnNextTip();
#0039     afx_msg HBRUSH OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor);
#0040     virtual void OnOK();
#0041     virtual BOOL OnInitDialog();

#0042     afx_msg void OnPaint();
#0043     //}}AFX_MSG
#0044     DECLARE_MESSAGE_MAP()
#0045
#0046     void GetNextTipString(CString& strNext);
#0047 };
#0048
#0049 #endif // !defined(TIPDLG_H_INCLUDED_)
```

TIPDLG.CPP（全新内容）

0001 #include "stdafx.h"

```
#0002  #include "resource.h"
#0003
#0004  // CG: This file added by 'Tip of the Day' component.
#0005
#0006  #include <winreg.h>
#0007  #include <sys\stat.h>
#0008  #include <sys\types.h>
#0009
#0010  #ifdef _DEBUG
#0011  #define new DEBUG_NEW
#0012  #undef THIS_FILE
#0013  static char THIS_FILE[] = __FILE__;
#0014  #endif
#0015
#0016  /////////////////////////////////////////////////////////////////////
#0017  // CTipDlg dialog
#0018
#0019  #define MAX_BUFLEN 1000
#0020
#0021  static const TCHAR szSection[] = _T("Tip");
#0022  static const TCHAR szIntFilePos[] = _T("FilePos");
#0023  static const TCHAR szTimeStamp[] = _T("TimeStamp");
#0024  static const TCHAR szIntStartup[] = _T("StartUp");
#0025
#0026  CTipDlg::CTipDlg(CWnd* pParent /*=NULL*/)
#0027         : CDialog(IDD_TIP, pParent)
#0028  {
#0029      //{{AFX_DATA_INIT(CTipDlg)
#0030      m_bStartup = TRUE;
#0031      //}}AFX_DATA_INIT
#0032
#0033      // We need to find out what the startup and file position parameters are
#0034      // If startup does not exist, we assume that the Tips on startup is checked TRUE.
#0035      CWinApp* pApp = AfxGetApp();
#0036      m_bStartup = !pApp->GetProfileInt(szSection, szIntStartup, 0);
#0037      UINT iFilePos = pApp->GetProfileInt(szSection, szIntFilePos, 0);
#0038
#0039      // Now try to open the tips file
#0040      m_pStream = fopen("tips.txt", "r");
#0041      if (m_pStream == NULL)
#0042      {
#0043            m_strTip.LoadString(CG_IDS_FILE_ABSENT);
#0044            return;
#0045      }
#0046
#0047      // If the timestamp in the INI file is different from the timestamp of
```

```
#0048      // the tips file, then we know that the tips file has been modified
#0049      // Reset the file position to 0 and write the latest timestamp to the
#0050      // ini file
#0051      struct _stat buf;
#0052      _fstat(_fileno(m_pStream), &buf);
#0053      CString strCurrentTime = ctime(&buf.st_ctime);
#0054      strCurrentTime.TrimRight();
#0055      CString strStoredTime =
#0056              pApp->GetProfileString(szSection, szTimeStamp, NULL);
#0057      if (strCurrentTime != strStoredTime)
#0058      {
#0059              iFilePos = 0;
#0060              pApp->WriteProfileString(szSection, szTimeStamp, strCurrentTime);
#0061      }
#0062
#0063      if (fseek(m_pStream, iFilePos, SEEK_SET) != 0)
#0064      {
#0065              AfxMessageBox(CG_IDP_FILE_CORRUPT);
#0066      }
#0067      else
#0068      {
#0069              GetNextTipString(m_strTip);
#0070      }
#0071  }
#0072
#0073  CTipDlg::~CTipDlg()
#0074  {
#0075      // This destructor is executed whether the user had pressed the escape key
#0076      // or clicked on the close button. If the user had pressed the escape key,
#0077      // it is still required to update the filepos in the ini file with the
#0078      // latest position so that we don't repeat the tips!
#0079
#0080      // But make sure the tips file existed in the first place....
#0081      if (m_pStream != NULL)
#0082      {
#0083              CWinApp* pApp = AfxGetApp();
#0084              pApp->WriteProfileInt(szSection, szIntFilePos, ftell(m_pStream));
#0085              fclose(m_pStream);
#0086      }
#0087  }
#0088
#0089  void CTipDlg::DoDataExchange(CDataExchange* pDX)
#0090  {
#0091      CDialog::DoDataExchange(pDX);
#0092      //{{AFX_DATA_MAP(CTipDlg)
#0093      DDX_Check(pDX, IDC_STARTUP, m_bStartup);
```

```
#0094     DDX_Text(pDX, IDC_TIPSTRING, m_strTip);
#0095     //}}AFX_DATA_MAP
#0096 }
#0097
#0098 BEGIN_MESSAGE_MAP(CTipDlg, CDialog)
#0099     //{{AFX_MSG_MAP(CTipDlg)
#0100     ON_BN_CLICKED(IDC_NEXTTIP, OnNextTip)
#0101     ON_WM_CTLCOLOR()
#0102     ON_WM_PAINT()
#0103     //}}AFX_MSG_MAP
#0104 END_MESSAGE_MAP()
#0105
#0106 /////////////////////////////////////////////////////////////////////////////
#0107 // CTipDlg message handlers
#0108
#0109 void CTipDlg::OnNextTip()
#0110 {
#0111     GetNextTipString(m_strTip);
#0112     UpdateData(FALSE);
#0113 }
#0114
#0115 void CTipDlg::GetNextTipString(CString& strNext)
#0116 {
#0117     LPTSTR lpsz = strNext.GetBuffer(MAX_BUFLEN);
#0118
#0119     // This routine identifies the next string that needs to be
#0120     // read from the tips file
#0121     BOOL bStop = FALSE;
#0122     while (!bStop)
#0123     {
#0124         if (_fgetts(lpsz, MAX_BUFLEN, m_pStream) == NULL)
#0125         {
#0126             // We have either reached EOF or enocuntered some problem
#0127             // In both cases reset the pointer to the beginning of the file
#0128             // This behavior is same as VC++ Tips file
#0129             if (fseek(m_pStream, 0, SEEK_SET) != 0)
#0130                     AfxMessageBox(CG_IDP_FILE_CORRUPT);
#0131         }
#0132         else
#0133         {
#0134             if (*lpsz != ' ' && *lpsz != '\t' &&
#0135                     *lpsz != '\n' && *lpsz != ';')
#0136             {
#0137                 // There should be no space at the beginning of the tip
#0138                 // This behavior is same as VC++ Tips file
#0139                 // Comment lines are ignored and they start with a semicolon
```

```
#0140                 bStop = TRUE;
#0141             }
#0142         }
#0143     }
#0144     strNext.ReleaseBuffer();
#0145 }
#0146
#0147 HBRUSH CTipDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
#0148 {
#0149     if (pWnd->GetDlgCtrlID() == IDC_TIPSTRING)
#0150         return (HBRUSH)GetStockObject(WHITE_BRUSH);
#0151
#0152     return CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
#0153 }
#0154
#0155 void CTipDlg::OnOK()
#0156 {
#0157     CDialog::OnOK();
#0158
#0159     // Update the startup information stored in the INI file
#0160         CWinApp* pApp = AfxGetApp();
#0161         pApp->WriteProfileInt(szSection, szIntStartup, !m_bStartup);
#0162 }
#0163
#0164 BOOL CTipDlg::OnInitDialog()
#0165 {
#0166     CDialog::OnInitDialog();
#0167
#0168     // If Tips file does not exist then disable NextTip
#0169     if (m_pStream == NULL)
#0170         GetDlgItem(IDC_NEXTTIP)->EnableWindow(FALSE);
#0171
#0172     return TRUE;  // return TRUE unless you set the focus to a control
#0173 }

#0174
#0175 void CTipDlg::OnPaint()
#0176 {
#0177     CPaintDC dc(this); // device context for painting
#0178
#0179     // Get paint area for the big static control
#0180     CWnd* pStatic = GetDlgItem(IDC_BULB);
#0181     CRect rect;
#0182     pStatic->GetWindowRect(&rect);
#0183     ScreenToClient(&rect);
#0184
#0185     // Paint the background white.
```

```
#0186      CBrush brush;
#0187      brush.CreateStockObject(WHITE_BRUSH);
#0188      dc.FillRect(rect, &brush);
#0189
#0190      // Load bitmap and get dimensions of the bitmap
#0191      CBitmap bmp;
#0192      bmp.LoadBitmap(IDB_LIGHTBULB);
#0193      BITMAP bmpInfo;
#0194      bmp.GetBitmap(&bmpInfo);
#0195
#0196      // Draw bitmap in top corner and validate only top portion of window
#0197      CDC dcTmp;
#0198      dcTmp.CreateCompatibleDC(&dc);
#0199      dcTmp.SelectObject(&bmp);
#0200      rect.bottom = bmpInfo.bmHeight + rect.top;
#0201      dc.BitBlt(rect.left, rect.top, rect.Width(), rect.Height(),
#0202             &dcTmp, 0, 0, SRCCOPY);
#0203
#0204      // Draw out "Did you know..." message next to the bitmap
#0205      CString strMessage;
#0206      strMessage.LoadString(CG_IDS_DIDYOUKNOW);
#0207      rect.left += bmpInfo.bmWidth;
#0208      dc.DrawText(strMessage, rect, DT_VCENTER | DT_SINGLELINE);
#0209
#0210      // Do not call CDialog::OnPaint() for painting messages
#0211   }
```

修改 ComTest 程序内容

以下是对于上述新增文件的分析与修改。稍早我曾分析过，只要修改一下 Splash Screen画面，增加一个TIPS.TXT 文字文件，再变化一下About 对话窗，就成了。

COMTEST.RC

要把自己准备的图片做为「炫耀画面」，有两个还算方便的作法。其一是直接编修Splash Screen 组件带给我们的Splsh16.bmp 的内容，其二是修改RC檔中的IDB_SPLASH 所对应的文件名称。我选择后者。所以我修改RC檔中的一行：

```
IDB_SPLASH  BITMAP  DISCARDABLE  "Dissect.bmp"
```

Dissect.bmp 图文件内容如下：



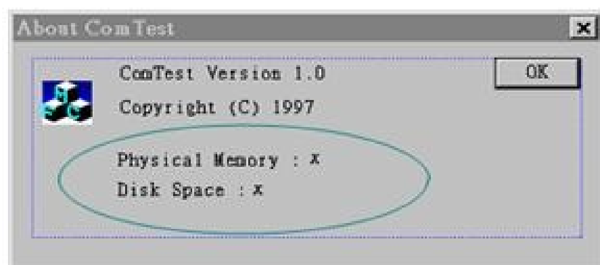
此外我也修改RC文件中的一些字符串，使它们呈现中文：

```
IDD_TIP DIALOG DISCARDABLE  0, 0, 231, 164
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "今日小秘诀"
FONT 8, "MS Sans Serif"
BEGIN
CONTROL       "",-1,"Static",SS_BLACKFRAME,12,11,207,123
LTEXT         "Some String",IDC_TIPSTRING,28,63,177,60
CONTROL       "程序启动时显示小秘诀",IDC_STARTUP,"Button",
BS_AUTOCHECKBOX | WS_GROUP | WS_TABSTOP,13,146,85,10
PUSHBUTTON    "下一个小秘诀",IDC_NEXTTIP,109,143,50,14,WS_GROUP
DEFPUSHBUTTON "关闭",IDOK,168,143,50,14,WS_GROUP
CONTROL       "",IDC_BULB,"Static",SS_BITMAP,20,17,190,111
END
STRINGTABLE DISCARDABLE
BEGIN
...
//  CG_IDS_DIDYOUKNOW     "Did You Know..."
CG_IDS_DIDYOUKNOW     "侯俊杰著作年表..."
END
```

增加一个TIPS.TXT

这很简单，使用任何一种文字编辑工具，遵循前面说过的 TIPS.TXT 文件格式，做出你的每日小秘诀。

修改RC檔中的About对话窗画面



我增加了四个static控制组件，其中两个做为标签使用，不必在乎其ID。另两个准备给ComTest程序在【About】对话窗出现时设定系统资讯使用，ID分 别设定为IDC_PHYSICAL_MEM和IDC_DISK_SPACE，配合System Info for About Dlg 组件的建议。

COMTEST.CPP

在CAboutDlg::OnInitDialog中利用SetDlgItemText 设定稍早我们为对话窗画面新增的两个static 控制组件的文字内容（Component Gallery 已经为我们做出这段程序代码，只是暂时把它标记为说明文字。我只要把标记符号//去除即可）：

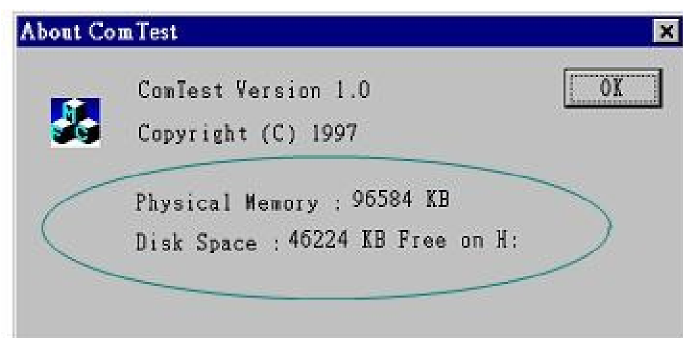
```
BOOL CAboutDlg::OnInitDialog()
{
        ...
        SetDlgItemText(IDC_PHYSICAL_MEM, strFreeMemory);
        ...
        SetDlgItemText(IDC_DISK_SPACE, strFreeDiskSpace);
    }
    return TRUE;    // CG:  This was added by System Info Component.
}
```

ComTest 修改结果

一切尽如人意。现在我们有了理想的 Splash Screen 画面如前所述，也有了 Tips of the Day 对话窗：



以及一个内含系统信息的 About 对话窗：



使用 ActiveX Controls

Microsoft的Visual Basic自1991年推出以来，已经成为Windows应用软件开发环境中的佼佼者。它的成功极大部份要归功于其开放性质：它所提供的 VBXs 被认为是一种极佳的面向对象程序设计架构。VBX 是一种动态链接函数库（DLL），类似Windows的订制型控制组件（custom control）。

VBX 不适用于32 位环境。于是Microsoft 再推出另一规格OCX。不论是 VBX 或OCX，或甚至 Borland 的VCL，都提供Properties-Method-Event（PME）接口。Visual Basic 之于VBX，以及 Borland C++ Builder 和 Delphi 之于 VCL，都提供了整合开发环境（IDE）与 PME 接口之间的极密切结合，使得程序设计更进一步到达「以拖拉、填单等简易动作就能够完成」的可视化境界。也因此没有人会反对把Visual Basic 和Delphi和C++ Builder 归类为RAD（Rapid Application Development，快速软件开发工具）的行列。但是Visual C++ 之于OCX，还没能够有这么好的整合。

我怎么会谈到OCX 呢？本节不是ActiveX Control 吗？噢，OCX 就是 ActiveX Control！由于微软把它所有的Internet 技术都称为ActiveX，所以 OLE Controls 就变成了ActiveX Controls。

我不打算讨论ActiveX Control 的撰写，我打算把全部篇幅用到ActiveX Control 的使用上。

如果对ActiveX Control 的开发感兴趣，Adam Denning 的ActiveX Control Inside Out是一本很不错的书（ActiveX 控制组件彻底研究，侯俊杰译/松岗）

ActiveX Control 基础观念：Properties、Methods、Events

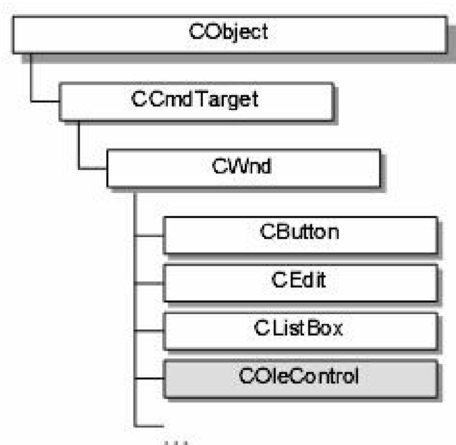
你必须了解ActiveX Control 三种接口的意义，并且充份了解你打算使用的某个 ActiveX Control 有些什么特殊的接口，然后才能够使用它。

基本上你可以拿你已经很熟悉的 C++ 类来比较 ActiveX control。类也是一个包装良好的组件，有它自己的成员变量，以及处理这些成员变量的所谓成员函数，是个自给自足的体系。ActiveX control 的三个接口也有类似性质：

- property - 相当于 C++ 类的成员变量
- method - 相当于 C++ 类的成员函数
- event - 相当于 Windows 控制组件发出的 notification 消息

ActiveX Control规格中定有一些标准的（库存的）接口，例如BackColor和FontName等properties，AddItem和Move和Refresh 等methods，以及CLICK 和KEYDOWN 等events。也就是说，任何一个ActiveX Control 大致上都会有一些必备的、基础的性质和能力。

以下针对ActiveX Control 的三种接口与C++ 类做个比较。至于它们的具体展现以及如何使用，稍后在实例中可以看到。



methods

设计自己的C++ 类，你当然可以在其中设计成员函数。此一函数之调用者必须在编译时期知道这一函数的功能以及它的参数。搭配Windows内建之控制组件（如 Edit、Button）而设计的类（如CEdit、CButton），内部固定会设计一些成员函数。某些成员函数（如CEdit::GetLineCount）只适用于特定类，但某些根类的成员函数（例如CWnd::GetDlgItemText）则适用于所有的子类。

ActiveX Control 的 method 极类似 C++ 类中的成员函数。但它们被限制在一个有限的集合之中，集合内的名单包括 AddItem、RemoveItem、Move 和 Refresh 等等。并不是所有的ActiveX Controls 都对每一个 method 产生反应，例如 Move 就不能够在每一个ActiveX Control 中运作自如。

properties

基本上properties用来表达 ActiveX Control 的属性或数据。一个名为 Date的组件可能会定义一个所谓的 DateValue，内放日期，这就表现了组件的数据。它还可能定义一个所谓的DateFormat，允许使用者取得或设定日期表现形式，这就表现了组件的属性。

你可以说ActiveX Control的properties 相当于C++ 类的成员变量。每一个 ActiveX Control可以定义属于它自己的properties，可以是一个字符串，可以是一个长整数，也可以是一个浮点数。有一组所谓的properties 标准集合（被称为 stock properties），内含BackColor、FontName、Caption 等等 properties，是每个ActiveX control 都会拥有的。

一般而言 properties 可分为四种类型：

- Ambient properties
- Extended properties
- Stock properties

- Custom properties

events

Windows 控制组件以所谓的notification（通告）消息送给其父窗口（通常是对话窗），例如按钮组件可能传送出一个BN_CLICKED。ActiveX Control 使用完全相同的方法，不过现在notification 消息被称为event，用来表示某种状况发生了。Events 的发射可以使ActiveX Control 有能力通知其宿主（container，也就是VB 或VC 程序），于是对方有机会处理。大部份 ActiveX Controls 送出标准的 events，例如 CLICK、KEYDOWN、KEYUP等等，某些 ActiveX Controls 会送出独一无二的消息（例如 ROWCOLCHANGE）。

一般而言 events 可分为两种类型：

- Stock events
- Custom events

ActiveX Controls 的五大使用步骤

欲在程序中加上 ActiveX Controls，基本上需要五个步骤：

1. 建立新项目时，在 AppWizard 的步骤 3 中选择【ActiveX Controls】。这会使程序代码多出一行：

```
BOOL COcxTestApp::InitInstance()
{
    AfxEnableControlContainer();
    ...
}
```

2. 进入Component Gallery，把ActiveX Controls 安插到你的程序中。

3. 使用ActiveX Controls。通常我们在对话窗中使用它。我们可以把资源编辑器的工具箱里头的 ActiveX Controls 拖放到目标对话窗中。

4. 利用ClassWizard 产生对话窗类，并处理相关的Message Maps、消息处理例程、变量定义、对话框函数等等。

5. 编译链接。

我将以系统内建（已注册过）的 Grid ActiveX Control 做为示范的对象。Grid 具有小型电子表格能力，当然它远比不上 Excel（不然 Excel 怎么卖），不过你至少可以获得一个中规中矩的7x14 电子表格，并且有基本的编辑和运算功能。

容我先解释我的目标。图16-1 是我期望的结果，这个电子表格完全为了家庭记账而量身设计，假设你有五种收入（真让人羡慕），这个表格可以让你登录每个月的每一种收入，并计

算月总收入和年总收入，以及各分项总收入。

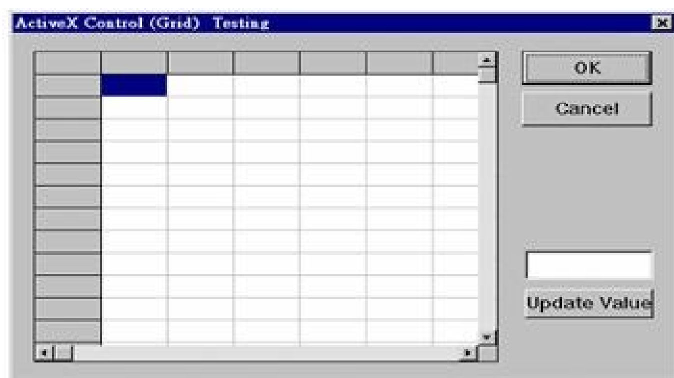


图16-1 在对话窗中使用Grid ActiveX control。

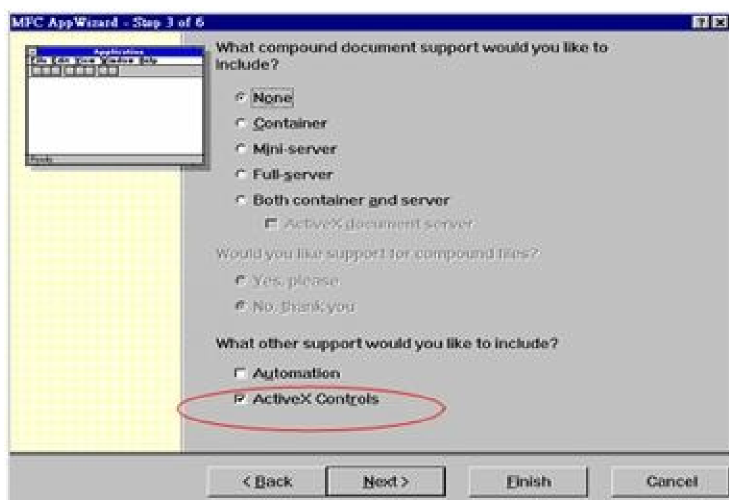
每一横列或纵行的最后一栏都是总和。

由于Grid 本身并不提供编辑能力，我们以电子表格右侧的一个edit字段做为编辑局部。使用者所选择的方格的内容会显示在这edit字段中，并且允许被编辑内容。数值填入后必须按下 `<Enter>` 键，或是在【Update Value】钮上按一下，电子表格内容才会更新。如果要直接在电子表格字段上做编辑动作，并不是不可以，把edit不偏不倚贴到字段也就是了!

本书进行到这里，我想你对于工具的使用应该已经娴熟了，我将假设你对于像「利用ClassWizard 为CMainFrame 拦 截一个ID_GridTest命令，并指名其处理常式为OnGridTest」这样的叙述，知道该怎么去动手。

使用**Grid ActiveX Control**：**OcxTest**程序

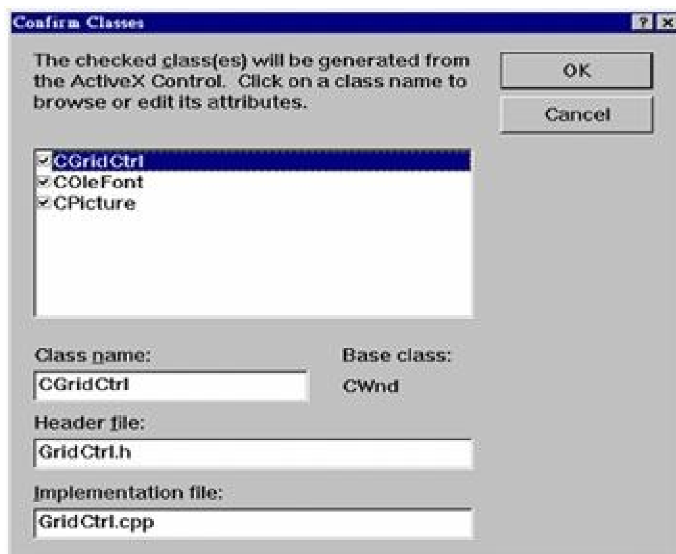
首先利用MFC AppWizard 做出一个OcxTest 项目。记得在步骤 3 选择【 ActiveX Controls】 ：



然后进入Component Gallery，将Grid安插到项目中：



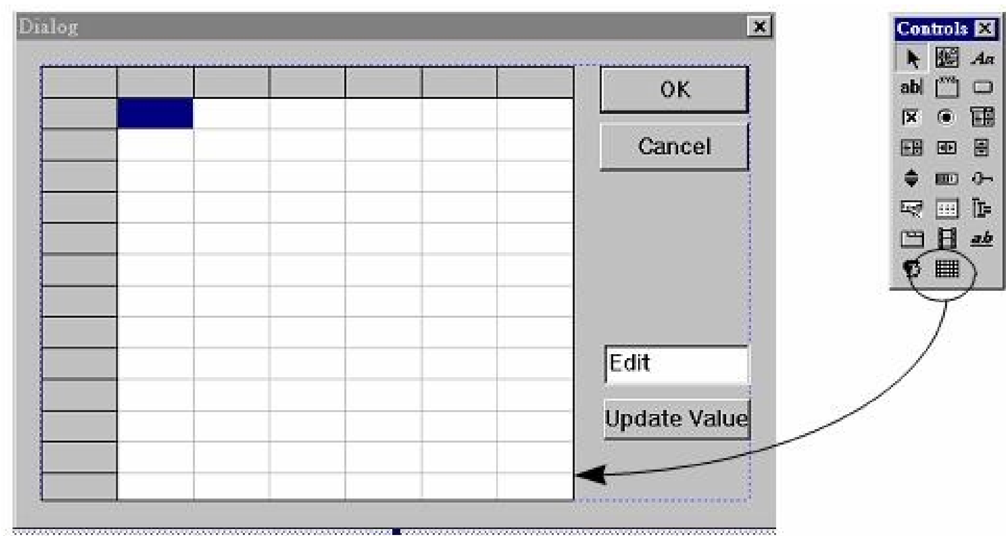
你必须回答一个对话窗：



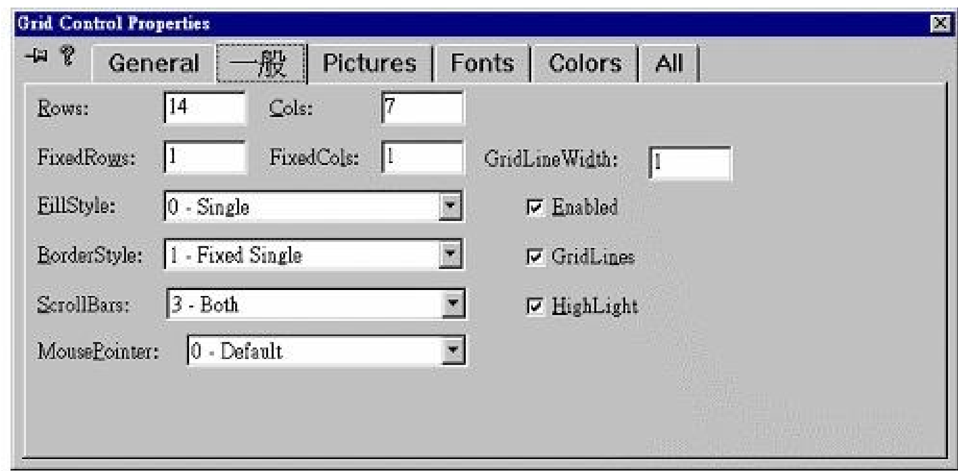
对话窗的设计

产生一个崭新的对话窗。这个动作与你在第10章为Scribble加上"Pen Width" 对话窗的步骤完全一样。请把新对话窗的ID从IDD_DIALOG1改变为 IDD_GRID。

从工具箱中抓出控制组件来，把对话窗布置如下。



虽然你把Grid 拉大，它却总是只有2x2 个方格。你必须使用右键把它的 Control Properties 引出来（如下），进入 Control 附页，这时候会出现各个 properties：



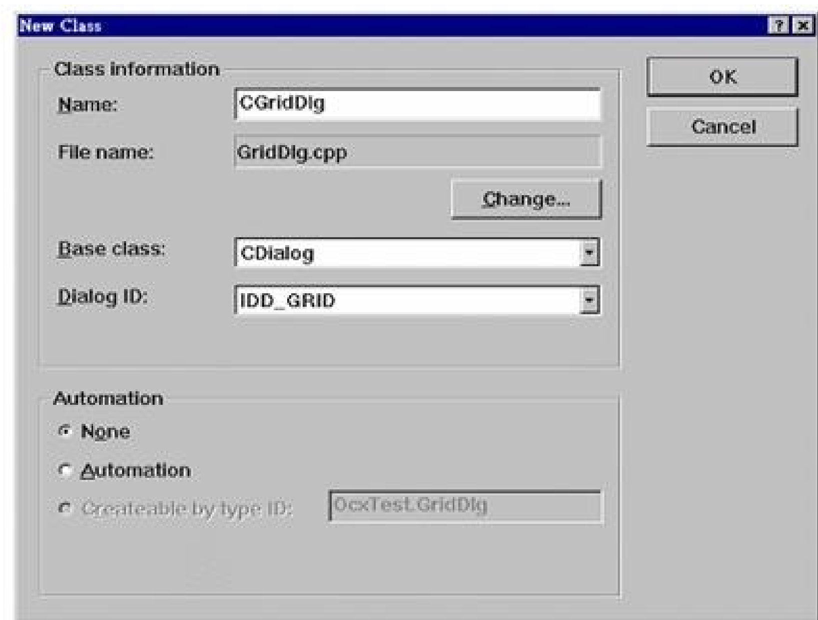
Control附页在中文Windows中竟然变成「一般」。这是否也算是一只臭虫？

现在选择Rows，设定为14，再选择Cols，设定为7。你还可以设定行的宽度和列的高度，以及方格初值...。噢，记得给这个 Grid 组件一个ID，叫做 IDC_GRID 好了。

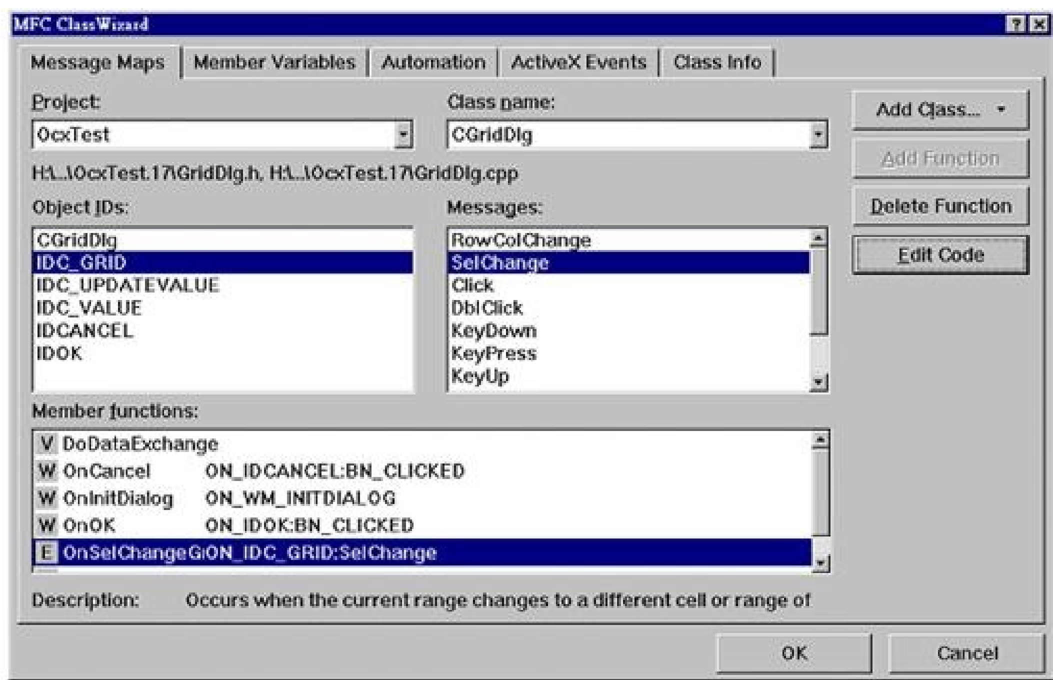
整个对话窗的设计规格如下：

```
对象                 ID                文字内容
对话窗               IDD_GRID         ActiveX Control (Grid) Testing
OK按钮              IDOK              OK
Cancel按钮          IDCANCEL          Cancel
Edit                  IDC_VALUE
Update Value按钮    IDC_UPDATEVALUE   Update Value
Grid                 IDC_GRID
```

现在准备设计IDD_GRID 的对话窗类。这件事我们在第10章也做过。进入CLassWizard，填写【Add Class】对话窗如下，然后按下【OK】钮：



回到ClassWizard主画面，准备为组件们设计消息处理例程。步骤是先选择一个组件ID，再选择一个消息，然后按下【Add Function】钮。注意，如果你选择到一个ActiveX Control，"Messages" 清单中列出的就是该组件所能发出的events。



本例的消息处理例程的设计规格如下：

对象ID	消息	处理函数名称

CGridDlg	WM_INITDIALOG	OnInitDialog
IDOK	BN_CLICK	OnOk
IDCANCEL	BN_CLICK	OnCancel
IDC_VALUE		
IDC_UPDATEVALUE	BN_CLICK	OnUpdatevalue
IDC_GRID	VBN_SELCHANGE	OnSelchangeGrid

到此为止，我们获得这些新文件：

```
RESOURCE.H
OCXTEST.RC
GRIDCTRL.H      <-- 本例不处理这个文件
GRIDCTRL.CPP    <-- 本例不处理这个文件
FONT.H          <-- 本例不处理这个文件
FONT.CPP        <-- 本例不处理这个文件
PICTURE.H       <-- 本例不处理这个文件
PICTURE.CPP     <-- 本例不处理这个文件
GRIDDLG.H       <-- 本例主要的修改对象
GRIDDLG.CPP     <-- 本例主要的修改对象
```

其中重要的相关程序代码我特别挑出来做个认识：

OCXTEST.RC

```
IDD_GRID DIALOG DISCARDABLE  0, 0, 224, 142
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "ActiveX Control (Grid) Testing"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,172,7,44,14
    PUSHBUTTON       "Cancel",IDCANCEL,172,24,44,14
    CONTROL          "",IDC_GRID,"{A8C3B720-0B5A-101B-B22E-00AA0037B2FC}",
                     WS_TABSTOP,7,7,157,128
    PUSHBUTTON       "Update Value",IDC_UPDATEVALUE,173,105,43,12
    EDITTEXT         IDC_VALUE,173,89,43,12,ES_AUTOHSCROLL
END
```

GRIDDLG.H

```
class CGridDlg : public CDialog
{
...
// Implementation
protected:

        // Generated message map functions
        //{{AFX_MSG(CGridDlg)
        virtual BOOL OnInitDialog();
        virtual void OnOK();
        virtual void OnCancel();
        afx_msg void OnUpdatevalue();
```

```
        afx_msg void OnSelChangeGrid();
        DECLARE_EVENTSINK_MAP()
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

GRIDDLG.CPP

```
BEGIN_MESSAGE_MAP(CGridDlg, CDialog)
        //{{AFX_MSG_MAP(CGridDlg)
        ON_BN_CLICKED(IDC_UPDATEVALUE, OnUpdatevalue)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////////////////////////////////////////////////////
/////
// CGridDlg message handlers

BEGIN_EVENTSINK_MAP(CGridDlg, CDialog)
    //{{AFX_EVENTSINK_MAP(CGridDlg)
    ON_EVENT(CGridDlg, IDC_GRID, 2 /* SelChange */, OnSelChangeGrid, VTS_NONE)
    //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

BOOL CGridDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here

        return TRUE;  // return TRUE unless you set the focus to a control
                      // EXCEPTION: OCX Property Pages should return FALSE
}

void CGridDlg::OnOK()
{
        // TODO: Add extra validation here

        CDialog::OnOK();
}

void CGridDlg::OnCancel()
{
        // TODO: Add extra cleanup here

        CDialog::OnCancel();
```

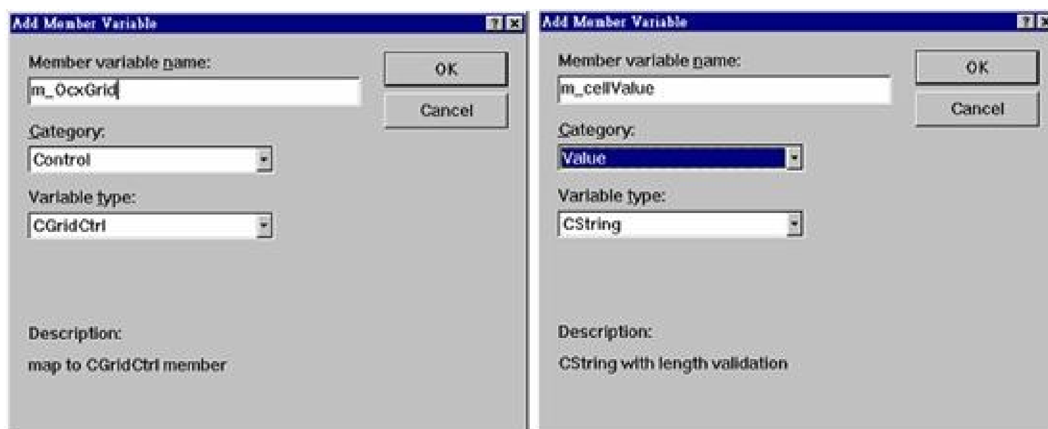
```
}

void CGridDlg::OnUpdatevalue()
{
        // TODO: Add your control notification handler code here
}

void CGridDlg::OnSelChangeGrid()
{
        // TODO: Add your control notification handler code here
}
```

为对话框加上一些变量

进入ClassWizard，进入【Member Variables】附页，选按其中的【Add Variable】钮，为OcxTest 加上两笔成员变量。其中一笔用来储存目前被选中的电子表格方格内容，另一笔数据用来做为 Grid 对象，其变量类型是 CGridCtrl：



这两个动作为我们带来这样的程序代码：

GRIDDLG.H

```
class CGridDlg : public CDialog
{
    // Dialog Data
    //{{AFX_DATA(CGridDlg)
    enum { IDD = IDD_GRID };
    CGridCtrl m_OcxGrid;
    CString m_cellValue;
    //}}AFX_DATA
    ...
};
```

GRIDDLG.CPP

```
CGridDlg::CGridDlg(CWnd* pParent /*=NULL*/)
: CDialog(CGridDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CGridDlg)
    m_cellValue = _T("");
    //}}AFX_DATA_INIT
}
void CGridDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CGridDlg)
    DDX_Control(pDX, IDC_GRID, m_OcxGrid);
    DDX_Text(pDX, IDC_VALUE, m_cellValue);
    //}}AFX_DATA_MAP
}
```

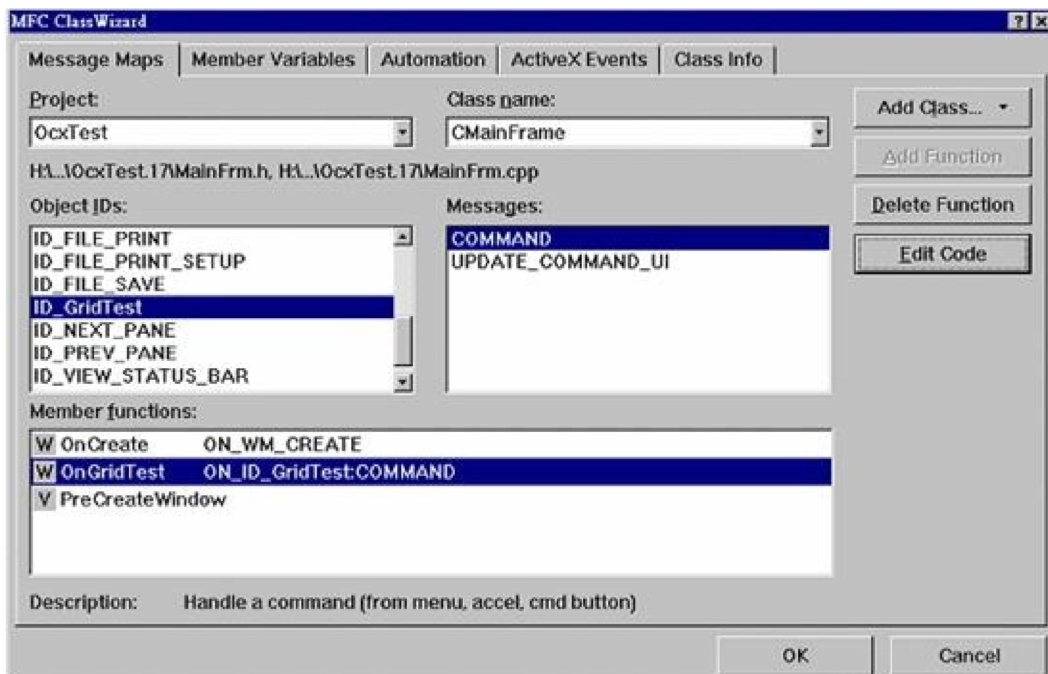
新增一个选单项目

利用资源编辑器，将选单修改如下：



注意，我所改变的选单是IDR_MAINFRAME，这是在没有任何子窗口存在时才会出现的选单。所以如果你要执行 OcxTest 并看到 Grid 组件，你必须先将所有的子窗口关闭。

现在利用 ClassWizard 在主窗口的消息映射表中拦截它的命令消息：



获得对应的程序代码如下：

MAINFRM.H

```
class CMainFrame : public CMDIFrameWnd
{
...
protected:
        //{{AFX_MSG(CMainFrame)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnGridTest();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

MAINFRM.CPP

```
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
        //{{AFX_MSG_MAP(CMainFrame)
        ON_WM_CREATE()
        ON_COMMAND(ID_GridTest, OnGridTest)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
void CMainFrame::OnGridTest()
{
        // TODO:
}
```

为了让这个新增选单命令真正发挥效用，将 Grid 对话窗唤起，我在 OnGridTest 函数加两行：

```
#include "GridDlg.h"
...
void CMainFrame::OnGridTest()
{
   CGridDlg dlg;           // constructs the dialog
   dlg.DoModal();          // starts the dialog
}
```

现在，将OcxTest编译连结一遍，得到一个可以顺利执行的程序，但Grid 之中全无内容。

Grid 相关程序设计

现在我要开始设计Grid 相关函数。我的主要的工作是：

准备一个二维（7x14）的 DWORD 数组，用来储存 Grid 的方格内容。

- 程序初始化时就把二维数组的初值设定好（本例不进行文件读写），并产生 Grid对话框。
- 对话框一出现，程序立刻把电子表格的行、列、宽、高，以及字段名称都设定好，并且把二维数组的数值放到对应方格中。初值的总和也一并计算出来。

- 把计算每一列每一行总和的工作独立出来，成立一个ComputeSums 函数。为了放置电子表格内容，必须设计一个7x14 二维数组。虽然电子表格中某些方格（如列标题或行标题）不必有内容，不过为求简化，还是完全配合电子表格的大小来设计数值数组好了。注意，不能把这个变量放在 AFX_DATA 之内，因为我并非以 ClassWizard 加入此变量。

GRIDDLG.H

```
#define MAXCOL 7
#define MAXROW 14

class CGridDlg : public CDialog
{
...
// Dialog Data
        double m_dArray[MAXCOL][MAXROW];

private:
        void ComputeSums();
};
```

为了设定Grid 中的表头以及初值，我在OnInitDialog 中先以一个for loop设定横列表头再以一个for loop 设定纵行表头，最后再以巢状（两层）for loop 设定每一个方格内容，然后才调用 ComputeSums 计算总和。

当使用者选择一个方格，其值就被 OnSelchangeGrid 拷贝一份到 edit 字段中，这时候就可以开始输入了。

OnUpdatevalue（【Update Value】按钮的处理例程）有两个主要任务，一是把edit字段内容转化为数值放到目前被选择的方格上，一是修正总和。

OnOk 必须能够把每一个方格内容（一个字符串）取出，利用atof转换为数值，然后储存到 m_dArray二维数组中。

GRIDDLG.CPP

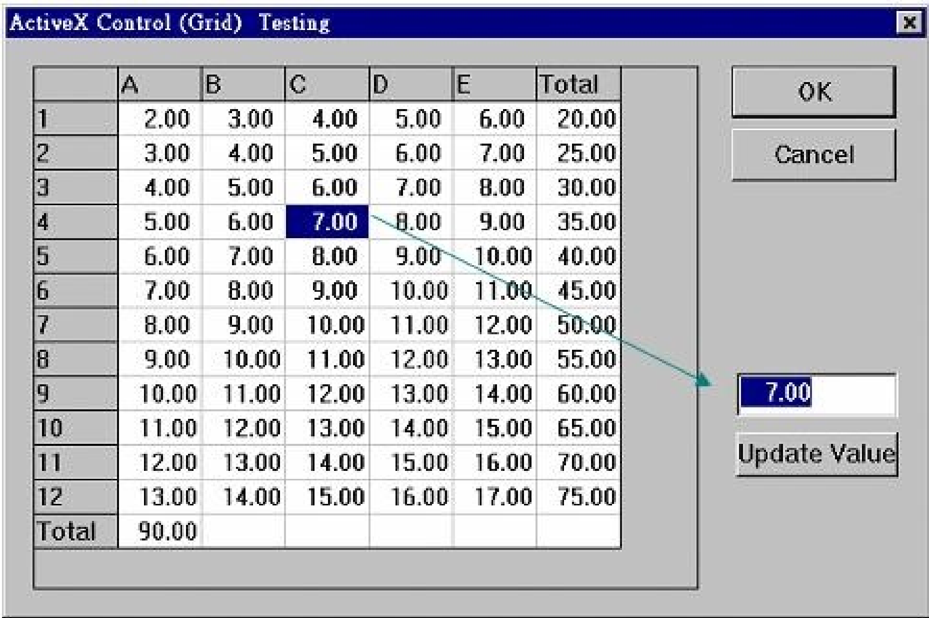
```
#0001
#0002  BOOL CGridDlg::OnInitDialog()
#0003  {
#0004      CString str;
#0005      int   i, j;
```

```
#0006      CRect rect;
#0007
#0008      CDialog::OnInitDialog();
#0009
#0010      VERIFY(m_OcxGrid.GetCols() == (long)MAXCOL);
#0011      VERIFY(m_OcxGrid.GetRows() == (long)MAXROW);
#0012
#0013      m_OcxGrid.SetRow(0);            // #0 Row
#0014      for (i = 0; i < MAXCOL; i++) {   // 所有的 Cols
#0015          if (i) { // column headings
#0016              m_OcxGrid.SetCol(i);
#0017              if (i == (MAXCOL-1))
#0018                  m_OcxGrid.SetText(CString("Total"));
#0019              else
#0020                  m_OcxGrid.SetText(CString('A' + i - 1));
#0021          }
#0022      }
#0023
#0024      m_OcxGrid.SetCol(0);            // #0 Col
#0025      for (j = 0; j < MAXROW; j++) {   // 所有的 Rows
#0026          if (j) { // row headings
#0027              m_OcxGrid.SetRow(j);
#0028              if (j == (MAXROW-1))
#0029                  m_OcxGrid.SetText(CString("Total"));
#0030              else {
#0031                  str.Format("%d", j);
#0032                  m_OcxGrid.SetText(str);
#0033              }
#0034          }
#0035      }
#0036
#0037      // sets the spreadsheet values from m_dArray
#0038      for (i = 1; i < (MAXCOL-1); i++) {
#0039          m_OcxGrid.SetCol(i);
#0040          for (j = 1; j < (MAXROW-1); j++) {
#0041              m_OcxGrid.SetRow(j);
#0042              str.Format("%8.2f", m_dArray[i][j]);
#0043              m_OcxGrid.SetText(str);
#0044          }
#0045      }
#0046
#0047      ComputeSums();
#0048
#0049      // be sure there's a selected cell
#0050      m_OcxGrid.SetCol(1);
#0051      m_OcxGrid.SetRow(1);
```

```
#0052     m_cellValue = m_OcxGrid.GetText();
#0053     UpdateData(FALSE); // calls DoDataExchange to update edit control
#0054     return TRUE;
#0055  }
#0056
#0057  void CGridDlg::OnOK()
#0058  {
#0059     int i, j;
#0060
#0061     for (i = 1; i < (MAXCOL-1); i++) {
#0062        m_OcxGrid.SetCol(i);
#0063        for (j = 1; j < (MAXROW-1); j++) {
#0064           m_OcxGrid.SetRow(j);
#0065           m_dArray[i][j] = atof(m_OcxGrid.GetText());
#0066        }
#0067     }
#0068     CDialog::OnOK();
#0069  }
#0070
#0071  void CGridDlg::OnUpdatevalue()
#0072  {
#0073     CString str;
#0074     double value;
#0075     // LONG   lRow, lCol;
#0076     int   Row, Col;
#0077
#0078     if (m_OcxGrid.GetCellSelected() == 0) {
#0079        AfxMessageBox("No cell selected");
#0080        return;
#0081     }
#0082
#0083     UpdateData(TRUE);
#0084     value = atof(m_cellValue);
#0085     str.Format("%8.2f", value);
#0086
#0087     // saves current cell selection
#0088     Col = m_OcxGrid.GetCol();
#0089     Row = m_OcxGrid.GetRow();
#0090
#0091     m_OcxGrid.SetText(str); // copies new value to
#0092                             //  the selected cell
#0093     ComputeSums();
#0094
#0095     // restores current cell selection
#0096     m_OcxGrid.SetCol(Col);
#0097     m_OcxGrid.SetRow(Row);
```

```
#0098  }
#0099
#0100  void CGridDlg::OnSelChangeGrid()
#0101  {
#0102      if (m_OcxGrid) {
#0103          m_cellValue = m_OcxGrid.GetText();
#0104          UpdateData(FALSE); // calls DoDataExchange to update edit
control
#0105          GotoDlgCtrl(GetDlgItem(IDC_VALUE)); // position edit control
#0106      }
#0107  }
#0108
#0109  void CGridDlg::ComputeSums()
#0110  {
#0111      int   i, j, nRows, nCols;
#0112      double sum;
#0113      CString str;
#0114
#0115      // adds up each row and puts the sum in the right col
#0116      // col count could have been changed by add row/delete row
#0117      nCols = (int) m_OcxGrid.GetCols();
#0118      for (j = 1; j < (MAXROW-1); j++) {
#0119          m_OcxGrid.SetRow(j);
#0120          sum = 0.0;
#0121          for (i = 1; i < nCols - 1; i++) {
#0122              m_OcxGrid.SetCol(i);
#0123              sum += atof(m_OcxGrid.GetText());
#0124          }
#0125          str.Format("%8.2f", sum);
#0126          m_OcxGrid.SetCol(nCols - 1);
#0127          m_OcxGrid.SetText(str);
#0128      }
#0129
#0130      // adds up each column and puts the sum in the bottom row
#0131      // row count could have been changed by add row/delete row
#0132      nRows = (int) m_OcxGrid.GetRows();
#0133      for (i = 1; i < MAXCOL; i++) {
#0134          m_OcxGrid.SetCol(i);
#0135          sum = 0.0;
#0136          for (j = 1; j < nRows - 1; j++) {
#0137              m_OcxGrid.SetRow(j);
#0138              sum += atof(m_OcxGrid.GetText());
#0139          }
#0140          str.Format("%8.2f", sum);
#0141          m_OcxGrid.SetRow(nRows - 1);
#0142          m_OcxGrid.SetText(str);
#0143      }
#0144  }
```

下图是OcxTest 的执行画面。


ActiveX Control (Grid) Testing
A B C D E Total
1 2.00 3.00 4.00 5.00 6.00 20.00
2 3.00 4.00 5.00 6.00 7.00 25.00
3 4.00 5.00 6.00 7.00 8.00 30.00
4 5.00 6.00 7.00 8.00 9.00 35.00
5 6.00 7.00 8.00 9.00 10.00 40.00
6 7.00 8.00 9.00 10.00 11.00 45.00
7 8.00 9.00 10.00 11.00 12.00 50.00
8 9.00 10.00 11.00 12.00 13.00 55.00
9 10.00 11.00 12.00 13.00 14.00 60.00
10 11.00 12.00 13.00 14.00 15.00 65.00
11 12.00 13.00 14.00 15.00 16.00 70.00
12 13.00 14.00 15.00 16.00 17.00 75.00
Total 90.00
OK
Cancel
7.00
Update Value

第五篇 附录

附录 A 无责任书评

从摇篮到坟墓 Windows 的完全学习

侯捷/1996.08.12整理

侯俊杰先生邀请我为他呕心沥血的新作深入浅出MFC写点东西。我未写文章久矣，但是你知道，要拒绝一个和你住在同一个大脑同一个躯壳的人日日夜夜旦旦夕夕的请求，是很困难的。不，简直是不可能。于是，我只好重作冯妇！

事实上也不全然是因为躲不过日日夜夜的轰炸，一部份原因是，当初我还在杂志上主持无责任书评时，就有读者来信希望书评偶而变换口味，其中一个建议就是谈谈如何养成Windows程序设计的全面性技术。说到全面性，那又是一个 impossible mission！真的，Windows 程序技术的领域实在是太广了，我们从来不会说游戏软件设计、多媒体程序设计、通讯软件设计... 是属于DOS程序技术的范畴，但，它们通常都被理所当然地归类属于Windows 程序设计领域。为什么？因为几乎所有的题目都拜倒在 Windows 作业系统的大伞之下，几乎每一种技术都被涵盖在千百计（并且以惊人速度继续增加中）的Windows API 之中。

我的才智实不足以涵盖这么大面积的学问，更遑论从中精挑细选经典之作介绍给你。那么，本文题目大剌剌的「完全学习」又怎么说？呃，我指的是 Windows 操作系统的核心观念以及程序设计的本质学能这一路，至于游戏、多媒体、通讯、Web Server、数据库、统统被我归类为「应用」领域。而Visual Basic、Delphi、Java 虽也都可以开发 Windows程序，却又被我屏弃在C/C++ 的主流之外。

以下谨就我的视野，分门别类地把我心目中认为必备的相关好书介绍出来。你很容易就可以从我所列出的书名中看出我的浅薄：在操作系统方面，我只涉猎 Windows 3.1和Windows 95（Windows NT 4.0是我的下一波焦点），在 Application Framework 方面，我只涉猎 MFC（OWL 和 Java 是我的下一个猎物）。

Windows 操作系统

Windows Internals / Matt Pietrek / Addison Wesley

最能够反应操作系统奥秘的，就是操作系统内部数据结构以及 API 的内部动作了。本书借着对这两部份所做的逆向工程，剖析Windows 的核心。

一个设计良好的应用程序接口（API）应该是一个不必让程序员担心的黑盒子。本书的主要立意并不在为了对API运作原理的讨论而获得更多程序写作方面的利益（虽然那其实是个必然的额外收获），而是藉由API伪代码，揭露出Windows 操作系统的运作原理。时光渐渐过去，程序员渐渐成长，我们开始对How感到不足而想知道Why了，这就是本书要给我们的东西。

本书不谈Windows官方手册上已有的信息，它谈「新信息」。如何才能获得手册上没有记载的信息？呵，原始代码说明一切。看原始代码当然是不错，问题是 Windows 的原始代码刻正锁在美国WA,Redmond（微软公司总部所在地）的保险库里，搞不好就在比尔.盖兹的桌下。我们唯一能够取得的Windows 原始代码大概只是SDK 磁盘上的defwnd.c 和defdlg.c（这是DefWindowProc 和DefDlgProc 的原始代码），以及 DDK 磁盘中的一大堆驱动程序原始代码。那么作者如何获得比你我更多的秘密呢？

Matt Pietrek 是软件反组译逆向工程的个中翘楚。本书藉由一个他自己开发的反组译工具，把获得的结果再以C虚拟代码表现出来。我们在书中看到许许多多的Windows API伪代码都是这么来的。Pietrek 还有一个很有名的产品叫做BoundsChecker，和SOFT- ICE/W（功能强大的Windows Debugger，以企鹅为形象）搭配销售。

本书主要探讨Windows 3.1 386加强模式，必要时也会提及标准模式以及 Windows 3.0。书中并没有涵盖虚拟驱动程序、虚拟机器、网络 API、多媒体、DDE/OLE、dialog/control等主题，而是集中在 Windows 启动程序、内存管理系统、窗口管理系统、消息管理系统、排程管理系统、绘图系统身上。本书对读者有三大要求：

- 对Intel CPU 的保护模式寻址方式、segmentation、selector 已有基本认识。
- 拥有Windows SDK 手册。
- 对操作系统有基础观念，例如什么是多任务，什么是虚拟内存...等等。

作者常借用面向对象的观念解释 Windows，如果你懂 C++ 语言，知道类与对象，知道成员函数和成员变量的意义与其精神，对他的比喻当能心领神会。

对系统感兴趣的人，本书一定让你如鱼得水。你唯一可能的抱怨就是：一大堆 API 函数的伪代码令人心烦气燥。文字瀚海图片沙漠的情形也一再考验读者的定力与耐力。然而小瑕不掩大瑜。我向来认为酿了一瓶好酒的人不必声嘶力竭地广告它，这本书就是一瓶好酒。作者Pietrek自1993/10 起已登上Microsoft Systems Journal 的Windows Q&A 主持人宝座，没两把刷子的人上这位子可是如坐针毡。现在他又主持同一本刊物的另一个专栏：Under The Hood。Dr. Dobb's Journal 的 Undocumented Corner 专栏也时有 Pietrek的踪影。

Undocumented Windows/Andrew Schulman,David Maxey,Matt Pietrek/ Addison Wesley